(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 December 2000 (14.12.2000)

PCT

(10) International Publication Number
WO 00/75849 A2

(51) International Patent Classification[7]: G06F 17/60

(21) International Application Number: PCT/US00/04249

(22) International Filing Date: 18 February 2000 (18.02.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/328,049 8 June 1999 (08.06.1999) US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 09/328,049 (CIP)
Filed on 8 June 1999 (08.06.1999)

(71) Applicant *(for all designated States except US)*: **BRIO TECHNOLOGY, INC.** [US/US]; 3460 West Bayshore Road, Palo Alto, CA 94303 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **LACKEY, Richard, L.** [US/US]; 16901 Spencer Avenue, Los Gatos, CA 95032 (US). **YEDWAB, Gadi** [US/US]; 4256 Fir Avenue, Seal Beach, CA 90740 (US).

(74) Agents: **BASINSKI, Erwin, J.** et al.; Morrison & Foerster LLP, 755 Page Mill Road, Palo Alto, CA 94304-1018 (US).

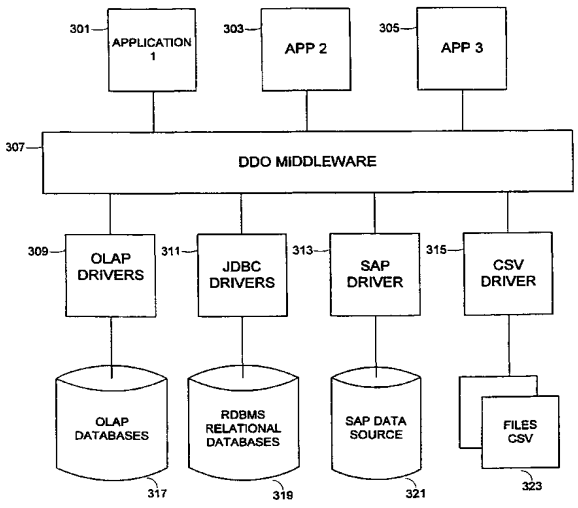
(81) Designated States *(national)*: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

**Published:**
— *Without international search report and to be republished upon receipt of that report.*

*[Continued on next page]*

(54) Title: METHOD AND APPARATUS FOR DATA ACCESS TO HETEROGENEOUS DATA SOURCES




(57) Abstract: The present invention is a middleware system which can provided efficient access to disparate data sources such as relational databases, non-relational databases, multidimensional databases and the like, in a manner which requires the selection of the data access parameters to be done only once and wherein these selected parameters are thereafter useable for the desired data access regardless of changes to the file structures of the data sources themselves. Additionally, access to newly specified data sources can be easily added to the system. Obtained data from the disparate data sources are displayed in a common format regardless of the source of the data and are displaced as streamed result sets.

WO 00/75849 A2

*For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.*

# METHOD AND APPARATUS FOR DATA ACCESS TO HETEROGENEOUS DATA SOURCES.

## TECHNICAL FIELD

This invention relates to the field of Computer Systems in the general Data Access and reporting sector. More specifically, the invention is a method and apparatus for a Middleware system which can provide easy access to data sources of disparate types.

## BACKGROUND ART

Enterprises have long pursued better business performance by investing in databases, data warehouses, Enterprise Reporting Procedure (ERP) systems, and other applications. These investments have been made in order to support the business management mantra that "If you can't measure it you can't manage it!" But in spite of all of their benefits, these systems are still failing to deliver on the promise of better business measurements and decisions. Today, the average employee wastes significant time trying to navigate the myriad systems in the typical enterprise to find the specific information they need, slowing decision making, hindering responsiveness, and even resulting in lost revenue opportunities. At the same time, a broader range of employees than ever before are demanding personalized access to enterprise information and they want to interact with that information in real-time, to help them make the "on-demand" decisions their jobs require. Many enterprises are now "extending" requirements for information access beyond employee end users to include suppliers, partners, and customers. The realities of this changing business context create mounting pressures for the Information Technology (IT) organization chartered with servicing the information needs of the extended enterprise. There is a need for a simplified system to permit users to quickly and easily access the data bases containing the Enterprise's information. Most especially this need exists for such systems to encompass the many disparate data base systems, such as manufacturing data and sales data, which may be in a relational data base, as well as personnel data which may be in a separate non-relational ERP data base, and other Enterprise data which may be contained in a multi-dimensional data base. Such a system must simplify access strategies

and reduce the cost of delivering timely business information to employees internally and to customers, partners, and suppliers over the Internet.

In the past, attempts have been made to address these problems. One such attempt was made by Sun Microsystems Inc.™ with its Java™ Database Connectivity (JDBC™) standard. JDBC provides a great interface for relational databases. However, JDBC makes two fundamental assumptions about its data sources. The first assumption is that the data source understand the Structured Query Language (SQL). The second assumption is that queries return "flat" data. JDBC is not equipped to address data sources that do not support a query language, and it does not support result-sets with complex structure such as objects, hierarchical and multidimensional data.

Another earlier attempt to solve the problem is made by the middleware products supplied by Sybase Inc.® Sybase separate products such as ClearConnect™, DirectConnect™, InfoHub™, jConnect™, OmniConnect™ provide data access frameworks to various kinds of heterogeneous data sources. However, in these products all data collected by these servers is relational, i.e., two dimensional, regardless of its native form. Thus an XML document or an ODBMS object will probably not be reflected in its native form. That is, because some simple structures are regular enough to transform into a relational structure (a two dimensional structure) that can be fixed in a RDBMS schema. However, the majority of real-world entities will not be convertible. Furthermore, it is inefficient to mutate a tree structured entity to a relational form to be converted back to an object structure by the receiving application. So, like JDBC, these middleware applications can look at anything as long as it is relational. Making something relational is the responsibility of the driver.

Another earlier attempt to solve this problem is the suite of Enterprise Data Access (EDA) middleware products created and marketed by Information Builders Inc.™ However this suite of products suffers from the same limitations as those identified above from Sybase Inc. That is, in these products all data collected by these servers is relational, i.e., two dimensional, regardless of its native form. Thus an XML document or an ODBMS object will probably not be reflected in its native form. That is, because some simple structures are regular enough to transform into a relational structure (a two dimensional

structure) that can be fixed in a RDBMS schema. However, the majority of real-world entities will not be convertible.

Another earlier attempt to solve this problem is the OLE/DB™ middleware products created and marketed by Microsoft™ Corporation. However this suite of products falls short of allowing applications to access complex non-relational data ("non-flat") without prior knowledge of the specifics of the data source. In contrast, what is needed are data sources and data that are are completely self-describing. Moreover, also needed are fully described data access procedures that take complex arguments and return complex data, and data access methods that are consistent for local, remote, relational, multidimensional, and object data.

These attempts can generally be described as creating a "metadata" layer that sits between a user application and a data base, the metadata layer representing a user's view of the universe represented by the database. The metadata layer usually relates statically to specific data bases and schemas, etc., within those databases. Another such metadata system is described in U. S. Patent No. 5,555,404 assigned to Business Objects, S. A. Paris, France. This system, like the ones described above, is a static view of the database that is limited in terms of the number of types of data sources it can access. In this particular case, any such source must support the SQL command language. In addition these static systems will fail if the underlying tables change but the metadata universe is not updated.

There is a need in the art for a single middleware product that defines an open interface for data access at a high level of abstraction allowing business-intelligence applications to treat vastly different data sources uniformly. There is a further need for a system that obtains metadata directly from the data sources, not from an intermediary repository which must be synchronized with a data source. There is a further need for a middleware product that permits the dynamic construction of a driver interface to a newly defined data source and permits the application to see the initial rows of results data as they are made available by the data source (hereinafter termed "streamed result sets").

## SUMMARY OF THE INVENTION

The present invention overcomes the disadvantages of the above-described systems by providing a high-performance, adaptable system and method for data access to disparate types of data sources, and wherein the metadata structure can be specified at run time.

The present invention is a middleware system (hereinafter "DDO" or "DDO middleware") which can provide efficient access to disparate data sources such as relational databases, non-relational databases, multidimensional databases, objects and XML files, in a manner which requires the selection of the data access parameters to be done only once and wherein these selected parameters are thereafter useable for the desired data access regardless of changes to the file structures of the data sources themselves. This list of data sources is not meant to limit the data sources which may be accessed by the present invention, but merely to indicate those sources accessable in the preferred embodiment. Additionally, access to newly specified data sources can be easily added to the system. Obtained data from the disparate data sources are displayed in a common format regardless of the source of the data and are displayed as streamed result sets. No other known data access system can provide this scalability, access to disparate data sources and "write once" aspect of the access and reporting rules.

The present invention includes an apparatus comprising a computer system, and a middleware mechanism coupled to the computer system and configured to access a plurality of data sources of disparate type in response to input commands from a user of the computer system.

The present invention further includes a method for using a computer for accessing data from a plurality of disparate data sources by a single application.

In addition, a computer program product embedded in a computer readable medium is claimed, wherein the product includes a middleware code mechanism for accessing a plurality of data sources of disparate type from a single application.

Other embodiments of the present invention will become readily apparent to those skilled in these arts from the following detailed description, wherein is shown and described only the embodiments of the invention by way of illustration of the best mode known at this time for carrying out the invention. The invention is capable of other and different embodiments some of which may be described for illustrative purposes, and

several of the details are capable of modification in various obvious respects, all without departing from the spirit and scope of the present invention.

## BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the system and method of the present invention will be apparent from the following description in which:

**Figure 1** illustrates a typical Internet network configuration.

**Figure 2** illustrates a representative general purpose computer client configuration.

**Figure 3** illustrates a representative block diagram of a preferred embodiment of the present invention.

**Figure 4** illustrates a more detailed representation of a preferred embodiment of the present invention.

**Figure 5** illustrates a block diagram representation of a DDO middleware driver according to a preferred embodiment of the present invention.

**Figures 6A** and **6B** illustrate a flow-chart of the general preferred embodiment process for creating a DDO application.

**Figure 7** illustrates a flow-chart of the general preferred embodiment process for creating a DDO driver.

## DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus for data access to heterogeneous data sources is disclosed. In the following description for purposes of explanation, specific data and configurations are set forth in order to provide a thorough understanding of the present invention. In the presently preferred embodiment the DDO middleware system is described in terms of a client system containing the DDO middleware. However, it will be apparent to one skilled in these arts that the present invention may be practiced without the specific details, in various applications such as a thin client-server configuration, wherein the DDO middleware system may reside primarily in the server. In other instances, well-known systems and protocols are shown and described in diagrammatical or block diagram form in order not to obscure the present invention unnecessarily.

ADDITIONAL BACKGROUND INFORMATION

The concepts of Object Oriented Programming (OOP) are well known in the programming and computer utilization arts. Never-the-less some basic information on OOP is of use to the understanding of this invention.

OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the program. An object is a software package that contains both data and a collection of related structures and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

In general, OOP components are reusable software modules which present an interface that conforms to an object model and which are accessed at run-time through a component integration architecture. A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each others capabilities or functions. This is generally done by assuming a common component object model on which to build the architecture. It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

OOP allows the programmer to create an object that is a part of another object. OOP also allows creation of an object that "depends from" another object. The relationship between these objects is called inheritance. The ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. Some typical categories are as follows:

- Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control system.
- Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.
- An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.
- An object can represent user-defined data types such as time, angles, and complex numbers, or points on the plane.

Programming languages such as C++ fully support the OOP principles, such as encapsulation, inheritance, polymorphism, and composition-relationship. C++ is an OOP language that offers a fast, machine-executable code, and is suitable for both commercial-application and systems-programming projects.

The benefits of object classes can be summarized, as follows:

- Objects and their corresponding classes break down complex programming problems into many smaller, simpler problems.
- Encapsulation enforces data abstraction through the organization of data into small, independent objects that can communicate with each other. Encapsulation protects the data in an object from accidental damage, but allows other objects to interact with that data by calling the object's member functions and structures.
- Subclassing and inheritance make it possible to extend and modify objects through deriving new kinds of objects from the standard classes available in the system. Thus, new capabilities are created without having to start from scratch.
- Polymorphism and multiple inheritance make it possible for different programmers to mix and match characteristics of many different classes and create specialized objects that can still work with related objects in predictable ways.
- Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.
- Libraries of reusable classes are useful in many situations, but they also have some limitations. For example:

- Complexity. In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.
- Flow of control. A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The programmer has to decide which functions to call at what times for which kinds of objects.

Class libraries are very flexible. As programs grow more complex, more programmers are forced to reinvent basic solutions to basic problems over and over again. A relatively new extension of the class library concept is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus, windows, dialog boxes, and other standard user interface elements for personal computers.

Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. Today, most personal computer software accomplishes the interaction with the user by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code according to actions that the user performs. The programmer no longer determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing control in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more

complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

Thus, as is explained above, a framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., for menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

- Behavior versus protocol. Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.
- Call versus override. With a class library, the code the programmer uses instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together.
- Implementation versus design. With class libraries, programmers reuse only implementations, whereas with frameworks, they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved. A preferred embodiment of the invention utilizes XML to implement documents on the Internet together with a general-purpose secure communication protocol for a transport medium between the client and the data sources. XML stands for Extensible Markup Language. XML is a system for defining, validating, and sharing document formats. XML uses tags (for example emphasis for *emphasis*), to distinguish document structures, and attributes (for example, in <A HREF="http://www.xml.com/">, HREF is the attribute name, and http://www.xml.com/ is the attribute value) to encode extra document information. XML will look very familiar to those who know about the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML).

To date, Web development tools have been limited in their ability to create dynamic Web applications which span from client to server and interoperate with existing computing resources. Until recently, HTML has been the dominant technology used in development of Web-based solutions. XML has been designed to be a more efficient technology but many of the HTML problems remain such as:

- Poor performance;
- Restricted user interface capabilities;
- Can only produce static Web pages;
- Lack of interoperability with existing applications and data; and
- Inability to scale.

Sun Microsystem's Java language solves many of the client-side problems by:

- Improving performance on the client side;
- Enabling the creation of dynamic, real-time Web applications; and
- Providing the ability to create a wide variety of user interface components.

With Java, developers can create robust User Interface (UI) components. Custom "widgets" (e.g., real-time stock tickers, animated icons, etc.) can be created, and client-side performance is improved. Unlike XML and HTML, Java supports the notion of client-side

validation, offloading appropriate processing onto the client for improved performance. Using the above-mentioned custom UI components, dynamic Web pages can also be created.

Sun's Java language has emerged as an industry-recognized language for "programming the Internet." Java supports programming for the Internet in the form of platform-independent Java applets. Java applets are small, specialized applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g., simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible browser (e.g., Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on C++. Sun's Java literature states that Java is basically, "C++ with extensions from Objective C for more dynamic method resolution."

Another technology that provides similar function to JAVA is provided by Microsoft™ and ActiveX™ Technologies, to give developers and Web designers wherewithal to build dynamic content for the Internet and personal computers. ActiveX includes tools for developing animation, 3-D virtual reality, video and other multimedia content. The tools use Internet standards, work on multiple platforms, and are being supported by over 100 companies. The group's building blocks are called ActiveX Controls, small, fast components that enable developers to embed parts of software in hypertext markup language (HTML) pages. ActiveX Controls work with a variety of programming languages including Microsoft Visual C++™, Borland Delphi™, Microsoft Visual Basic™ programming system and, in the future, Microsoft's development tool for Java, code named "Jakarta." ActiveX Technologies also includes ActiveX Server Framework, allowing developers to create server applications. One of ordinary skill in the art readily recognizes that ActiveX could be substituted for JAVA without undue experimentation to practice the invention.

OPERATING ENVIRONMENT

The environment in which the present invention is used encompasses the general distributed computing scene which includes generally local area networks with hubs, routers, gateways, tunnel-servers, applications servers, etc. connected to other clients and other networks via the Internet, wherein programs and data are made available by various members of the system for execution and access by other members of the system. Some of the elements of a typical internet network configuration are shown in **Figure 1**, wherein a number of client machines **105** possibly in a branch office of an enterprise, are shown connected to a Gateway/hub/tunnel-server/etc. **106** which is itself connected to the internet **107** via some internet service provider (ISP) connection **108**. Also shown are other possible clients **101, 103** similarly connected to the internet **107** via an ISP connection **104**, with these units communicating to possibly a home office via an ISP connection **109** to a gateway/tunnel-server **110** which is connected **111** to various enterprise application servers **112, 113, 114** which could be connected through another hub/router **115** to various local clients **116, 117, 118**.

An embodiment of the DDO Middleware system can operate on a general purpose computer unit which typically includes generally the elements shown in **Figure 2**. The general purpose system **201** includes a motherboard **203** having thereon an input/output ("I/O") section **205**, one or more central processing units ("CPU") **207**, and a memory section **209** which may have a flash memory card **211** related to it. The I/O section **205** is connected to a keyboard **226,** other similar general purpose computer units **225, 215**, a disk storage unit **223** and a CD-ROM drive unit **217**. The CD-ROM drive unit **217** can read a CD-ROM medium **219** which typically contains programs **221** and other data. Logic circuits or other components of these programmed computers will perform series of specifically identified operations dictated by computer programs as described more fully below.

A preferred embodiment of the DDO middleware will be described in terms of a Workstation with Windows NT™ or Windows 95/98™, and the assumption is that one skilled in these arts understands Basic Java™ programming skills and has access to a Java programming tool such as Symantec™ Visual Cafe™, Visual J++™, Visual Age™ for

Java, or JBuilder™, in addition to the Software Development Kit for DDO ("DDO-SDK"). The DDO-SDK provides all the necessary tools and libraries to develop a driver for a data source, and is described in more detail below with regards to a preferred embodiment.

## THE INVENTION

The present invention (DDO) is a middleware system which provides an abstraction for heterogeneous data sources. While Structured Query Language (SQL), Open Data Base Connectivity (ODBC) and Java Data Base Connectivity (JDBC) are limited to relational data, DDO supports much broader data access. It defines an open interface for data access at a high level of abstraction allowing applications to treat vastly different data sources uniformly. DDO provides a superior Java alternative to Microsoft's ADO (ActiveX Data Object). And DDO provides a more efficient alternative to the Microsoft OLE/DB middleware products. This Microsoft suite of products does not allow applications to access complex non-relational data ("non-flat") without prior knowledge of the specifics of the data source. In contrast, DDO data sources and data are completely self-describing. Moreover, DDO can fully describe data access procedures that take complex arguments and return complex data. DDO data access methods are consistent for local, remote, relational, multidimensional, and object data.

In the preferred embodiment, DDO provides additional unique capabilities that are not found in OLE/DB or other existing middleware products. DDO provides a mechanism for applications to discover the required logon parameters. Most middleware products either assume that these parameters are limited to a user name and a password, or they require that the application would have prior knowledge of the parameters that the data source requires. By providing meta-data about logon attributes, DDO allows applications to connect to any data source without such prior knowledge or assumption. Because of the unique capabilities model, DDO can support a Query Builder that knows nothing about the data sources. This is facilitated by the capabilities, properties, meta-data, parameters, and local-language descriptions.

The capabilities model provides a mechanism for applications to learn (discern) data source attributes. The properties aspect of the model provides a means for applications

to tell DDO about their capabilities. This two-way model provides the basis for application/DDO driver negotiation.

Another capability of DDO is allowing data access drivers to be easily developed by simply extending a skeleton driver implementation. This allows developers to create drivers for their proprietary data sources in a matter of days, rather than months. This is an important capability that is absent in OLE/DB, JDBC, EDA and other such middleware products.

In the preferred embodiment, DDO places fewer expectations on the data source delivery model. A data source, for example, could be a real-time feed, e.g., stock ticker. The execution model supports streamed results for pipeline execution. This dramatically lowers the working set size for the DDO execution model. This is not generally true of other such middleware products.

DDO embraces ERP (Enterprise Resource Planing) and other multi-tier applications in which data is abstracted as business objects. Information access through business objects enforces application security and encapsulates business rules.

DDO is open. It makes no assumptions about the application and the data source. It provides an open interface for any query or reporting tool. Data sources can be relational as well as non-relational. DDO provides access to application business objects, multi-dimensional and hierarchical data, XML data, and arbitrarily complex data.

DDO provides access to metadata, allowing applications to interactively discover the information objects and the capabilities of the data source. Reporting applications can select data and build queries without intimate knowledge of the data source. They can make selections and build queries in a generic manner without having to separately support each data source.

In the preferred embodiment DDO includes a Software Development Kit (SDK) section making it easy to add a new data source by implementing a driver. The driver declares the capabilities and properties of the data source. Most capabilities are optional and the driver simply specifies which capabilities are supported.

DDO provides remote data access by marshalling data access requests to a DDO server and returning results back to the client. This allows the distribution of DDO drivers and clients if desired by a user.

A typical system of the preferred embodiment **300** is shown in **Figure 3** wherein the DDO middleware layer **307** is shown between a plurality of applications **301, 303** and **305,** the DDO middleware layer connected on the back end to a plurality of drivers **309, 311, 313, 315 and 316.**. These drivers, as explained in more detail below, provide the interface between the language and presentation rules of the DDO middleware engine **307** and the language and query rules (if any) of the respective data source **317, 319, 321, 323** and **324.**

## HOW TO MAKE AND USE THE INVENTION

DDO provides a Java interface that conforms to the Sun Microsystems JavaBeans™ specification. Access to DDO from other programming languages is provided using COM™ and CORBA™ object-access methods. Referring now to **Figure 4** some of the main interfaces of DDO are highlighted. In **Fig. 4** the DDO middleware layer **401** comprises a Transaction Interface **403** which controls the Begin, Commit and Rollback facilities **405**, a Data Source Manager **407** which makes use of a Registry File **409**, A Data Source Interface **411** which makes use of Property Sheets **413,** and a Connection Interface **415**. Also included in the DDO middleware layer **401** are a Metadata mechanism **417**, an Obtain/Get Data Mechanism **419**, a Process Results Mechanism **421** and a Driver Manager mechanism **423**. These are explained in more detail below.

### Data Source Manager 407

The Data Source Manager **407** keeps track of the available data sources in the system. It maintains a persistent registry **409** of data sources. When a new data source is registered, the name of the data source and the name of its implementation class (the driver) are saved. Multiple data sources that are of the same type can use the same driver.

### Data Source

The data source interface **411** describes the capabilities and properties of the data source and allows the application to create a connection for the data source. Data Sources can implement different query languages. Moreover, support for a query language is optional. A data source that does not support a query language may still support data filtering through a simple selector interface, giving the opportunity to reduce the amount of data that is retrieved.

### Connection

The connection interface **415** provides methods that obtain metadata and data. Metadata **417** describe the hierarchy of objects, the fields that the object provides, and parameters. Data can be obtained by executing a command, calling a procedure, or simply by naming an object and requesting its data.

### Transaction

The transaction interface **403** provides methods for beginning a transaction, commit, and rollback **405**. A data source is not required to support transactions. The corresponding capability indicates that this data source does not support transactions. The scope of a transaction is one data source. DDO does not coordinate transactions between multiple data sources.

### Command

A reporting application executes a command by passing a command text and associated parameters. These commands are processed in the DDO middleware layer by the "Obtain/Get Data Mechanism" **419.** The result is a Rowset which is processed by the "Process Results Mechanism" **421**. The command can be anything that the data source understands. A data source describes whether it can accept SQL and the level of SQL that it understands. This allows reporting applications to send standard SQL to any data source that understands SQL as well as to pass-through data-source specific commands.

### Procedure

This interface, which is part of the "Obtain/Get data mechanism" **419,** abstracts any parameterized object, remote procedure call, stored database procedure or any object that behaves like a procedure. An application specifies the procedure it wants to execute and supplies any IN and IN/OUT parameters. The result object may contain a status and any number of Rowset objects representing multiple result sets.

### Selector

This interface, which is also part of the "Obtain/Get data mechanism" **419,** is for the simplest data sources that typically do not support a command language. They simply

provide a set of BI objects. A DDO application uses a Selector to specify the fields that it wants from an object, as well as filtering and sorting.

In connection with the Procedure and Selector Interfaces, the Driver indicates what retrieval mechanism it supports, for example, command and selectors, and then the application can choose which mechanism it prefers to use. The mechanism support is indicated through Capabilities.

**Rowset, Row, and Call Results**

The Call Results interface **421** allows access to the results of a procedure call. This includes the return value, output parameters, and the Rowset objects that the procedure returns.

Rowset represents a forward-only read-only set of rows. It does not cache the entire set in memory. Once the next row is retrieved, the previous row may be discarded. A row may be retrieved from the database just in time for it to be retrieved. This allows the Rowset implementation to only keep a small number of rows in memory at any given time and therefore can handle very large results.

These concepts and mechanisms will become more clear through an example of a preferred embodiment which is described below.

Referring now to **Figure 5**, a DDO middleware Driver **500** is depicted. In **Fig. 5** a DDO driver **501** is shown comprising: Message logging and timing data tools **503**, a Capabilities resolution file **505**, DDO base classes **507** and an exception and localized error message framework **509**. The DDO driver **501** also contains the basic translators to translate the data structure of the particular data source into the standardized presentation form of the DDO system **511** and to translate commands from the standardized presentation form of the DDO system into the data structure of the particular data source **513**. These components will be described in more detail below.

Referring now to **Figure 6A** a general process of creating a DDO Application is depicted. In **Fig. 6A** a process begins by attempting to Locate a Data Source **603**. This attempt makes use of DataSourceManager to check the registry files **605** to see if the desired Data Source is registered. If the desired Data Source exists in the registry **611** then a pointer from the registry file points to the driver to use **617**. If the desired Data Source is

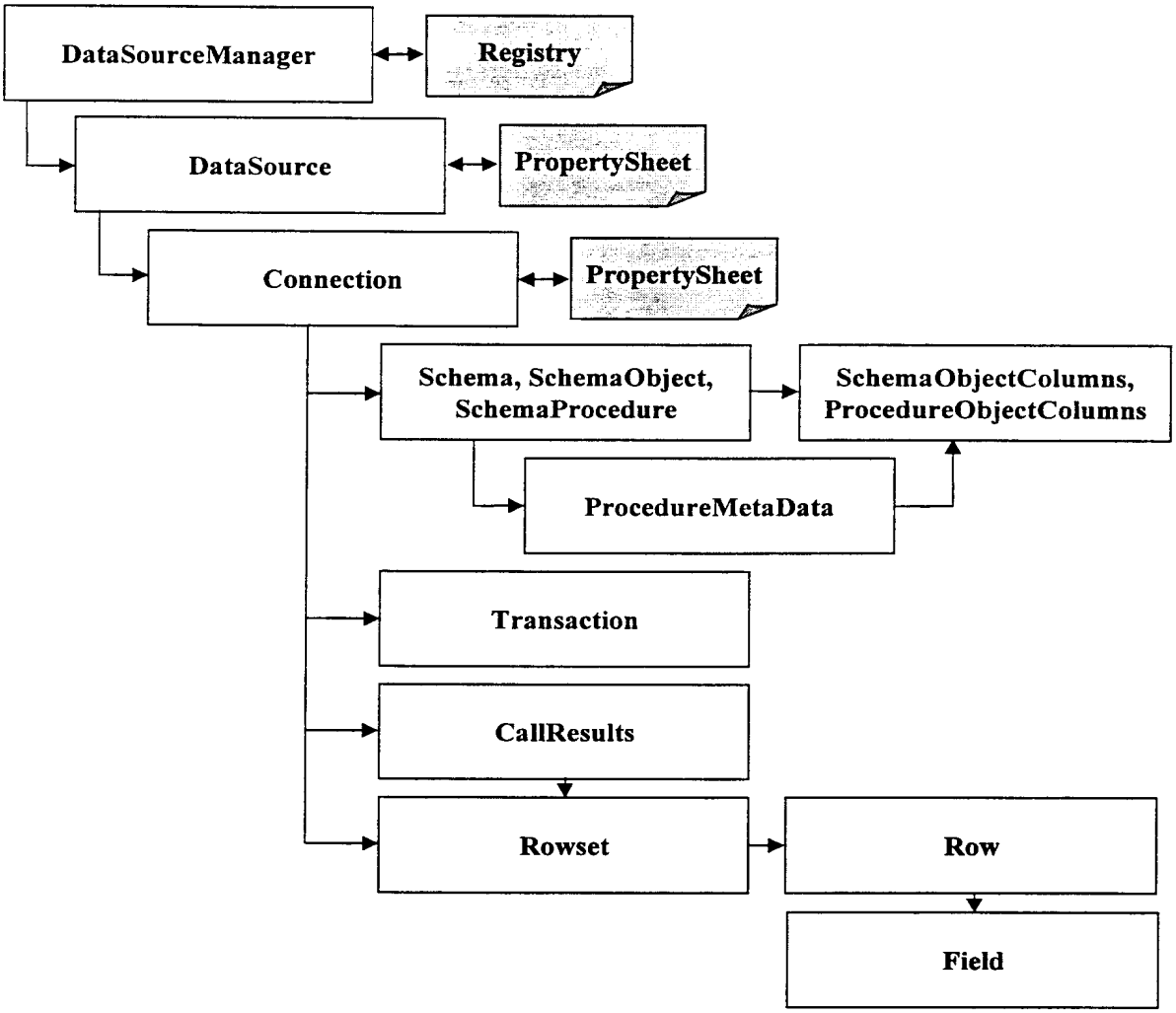
not found in the registry **609** then the new Data Source must be defined **613** and a new driver for this new data source must be registered **615**. Once a driver is obtained then a connection to the data source is established **619**, and Logon properties are processed **621**. Next capabilities (read-only properties that describe attributes of the data source) are discovered **623** and the DDO middleware layer proceeds to get Metadata **625**, get objects & procedures related to this data source **627**, and from these, commands to access data in this data source can be constructed. Commands are then executed against this data source **629**, the results processed **631** and the data displayed to the user **633**.

These steps in the process are now explained in more detail by way of an example of how this works in a preferred embodiment of the present invention.

### *The DDO API*

The DDO API defines Java interfaces that represent data sources, connections, information objects, result sets, and metadata. It allows applications to request data and process the results. The DDO API includes a driver manager that can support multiple drivers connecting to data sources. An application can enumerate the available data sources and create connections to them.

The diagram below depicts the flow of processing in a typical DDO application. The DataSourceManager (**407 in Fig. 4**) maintains lists of available data sources in Registries. Using the DataSourceManager class the application locates and loads a DataSource interface.




The DataSource interface **411** represents a data source. With the data source, a PropertySheet **413** describes the specific properties of this data source such as the properties that are required to open a connection to the Data source. Using the "open()" method of the DataSource interface, a connection to the data source is created and a Connection interface **415** is returned.

The Connection interface is the main interface in DDO. This interface is used to obtain metadata, obtain data, execute commands and procedures, and perform transactions.

The Schema, SchemaObject, and SchemaProcedure classes provide rich metadata that describe all the information objects that are available through this connection. The SchemaObjectColumns and SchemaProcedureColumns describe the data that is available in each information object.

If the data source supports transactions, then the Transaction interface provides methods for commit and rollback.

Data is obtained by requesting data from an information object. This can be done using the "getData()" method of Connection. The application names the information object from which data is requested and optionally specifies some selection criteria. The result is a Rowset interface. Another way of obtaining data is using the "execute()" method. This method sends a Command to the data source. The Command contains a statement that the data source can recognize along with any parameters. Results are returned via the Rowset interface.

A third method of obtaining data is by calling a procedure using the "call()" method of Connection. This returns a CallResults interface. This interface allows the application to obtain output parameters, a return value, and any number or rowsets via the Rowset interface.

The Rowset interface is the main interface for processing results in DDO. It provides methods that retrieve the results a row at a time. Each Row contains Field objects. These are typed objects that are derived from the abstract Field class. The Rowset, Row, and Field objects are all self-describing. They provide information about the number, type, size, and name of the data items that they hold.

In the following section a simple example is shown that uses DDO to retrieve data from the Employee table in the Accounting database.

### *A Simple Code Example*

The following code snippet demonstrates the ease in which a data source is located, a connection is created, and a data is retrieved. (The code omits some error checking. This is shown later.) Results are also easily processed. The example shows how to take advantage of the fact that Rowset is self-describing and obtain the names of the columns being returned. Fetching and listing the data is also shown. In the example the types of the fields being returned are not a concern at this time. They are all implicitly converted to text.

The DDO API is implemented in the Java package "com.sqribe.access".

```
import com.sqribe.access.*;
...
try {
   DataSource ds = DataSourceManager.getDataSource("accounting");
   ds.getPropertySheet().setProperty("user", "scott");
   ds.getPropertySheet().setProperty("password", "tiger");
   Connection c = ds.open();
   Rowset rs = c.getData(c.getSchemaObject("Employee"));
   System.out.println("<table><tr>");
   // print headings
   for (int i = 0; i < rs.getFieldCount(); i++)
      System.out.println("<td>" + rs.getField(i).getName() + "</td>");
   while (rs.next()) {
      // print data
      System.out.println("</tr><tr>");
      for (int i = 0; i < rs.getFieldCount(); i++)
         System.out.println("<td>" + rs.getField(i) + "</td>");
   }
   System.out.println("</tr></table>");
   c.close(); // close the connection to the data source
```

### *Managing Data Sources*

The DataSourceManager **411** maintains a list of available data sources called Registry **409**. Data sources are identified by their name. Using the Registry, the DataSourceManager can locate and instantiate a data source and return a DataSource interface to the application.

To locate a data source using its name and obtain a DataSource interface, an application calls the "DataSourceManager.getDataSource()" method. For example,

```
DataSource ds;
ds = DataSourceManager.getDataSource("accounting");
if (ds == null) {
   System.out.println("The accounting data source is unavailable.");
}
```

In this code example, the DataSourceManager searches the Registry for a data source called *accounting*. If found, the driver for this data source is loaded and its DataSource interface is returned. If the data source is not found in the Registry or the data source entry in the Registry is improperly configured, a null reference is returned. (Note that multiple calls to "getDataSource()" with the same name return the same object).

You can use the "DataSourceManage.getDataSources()" method to obtain a list of the available data sources in the current registry.

```
import com.sqribe.access.*;
import java.util.Enumeration;
...
Enumeration enum;
enum = DataSourceManager.getDataSources();
while (enum.hasMoreElements()) {
    DataSource ds = (DataSource)enum.nextElement();
    System.out.println(ds.getName());
}
```

By default, the registry contains entries that are stored in the file "Registry.properties". This file can be found in the "properties" folder.

In the next section, how to manage multiple registries is examined. However, if the application will always use the default registry (the file "Registry.properties"), then one can skip this section and proceed directly to the section "Defining a New Data Source".

**Registry Files**

The Registry **409** is a memory list of data sources maintained by the DataSourceManager **407**. When the DataSourceManager is used for the first time, it loads the Registry from a file. The default Registry file is "Registry.properties". This file is located in the DDO "properties" folder.

The application can load the Registry from alternative sources. This is done using the DataSourceAdmin class. This class provides static methods for loading DataSource definitions from a file or InputStream.

A registry file contains a list of registered data sources along with a short description for the data source, the name of its implementation class, and a connection string.

The same DataSource implementation class may be used to implement multiple data sources. Each data source is registered separately.

The following code example loads the Registry from a file called "MyDataSources.properties". This file is located in the "properties" folder.

```
try {
    DataSourceManager.getDataSourcesAdmin().load("MyDataSources");
} catch (DataAccessException e) {
    e.printStackTrace();
}
```

Note that the ".properties" extension of the registry file is assumed and should not be specified in the "load()" call. Also note that like most of DDO calls, this call may fail with a DataAccessException.

In addition to loading registry entries from a properties file in the "properties" folder, the DataSourcesAdmin class allows one to load registry entries from any file or InputStream. Multiple loads can be performed. The effect is cumulative. The DataSourcesAdmin class also allows one to save the current registry into a file or OutputStream.

#### The DataSourceManagerAdmin Class

The DataSourceManager uses the DataSourceManagerAdmin class to maintain its properties in a file named com_sqribe_access_DataSourceManager_Properties. This file is located in the DDO "properties" folder.

Within this properties file there is a property called "DataSources.files":

```
DataSources.files = <list of data sources property files>
```

For example,

```
DataSources.files = Registry MyDataSources
```

The DataSourceManagerAdmin allows one to enumerate these names and subsequently use them in "DataSourcesAdmin.load()" calls. This allows an application to deal with multiple registries. For example, one has a restricted registry with only a few data sources and a power user's registry with all available data sources. The class also allows an administrative tool to obtain the list of useable registries.

### *Defining a New Data Source*

A data source is defined in a registry file. You can either use the tool that's supplied with the DDO SDK or edit the registry file directly.

#### Creating a New Data Source by Editing a Registry File

To understand how to add a new data source to a registry file, consider the following example. In this example, we will add a data source called HelpDesk to the default registry file, "Registry.properties". The HelpDesk data source is an Oracle database. To access this database we will make use of the DDO JDBC Access driver and use a JDBC driver from Oracle.

The following steps will be used:

- Identify the packages that must be added to the CLASSPATH.
- Edit the "Registry.properties" file.

- Specify the name of the data source.
- Specify the DDO driver.
- Specify loading of the JDBC driver.
- Specify the JDBC URL.

**Identify the Packages That Must Be Added to the CLASSPATH**

To access a database via JDBC we will use the JDBC Access DDO driver. This driver is implemented in the package "com.sqribe.jdbcacc" and distributed in the file "jdbcacc.jar". If this JAR file is not already part of your CLASSPATH environment variable, then add it to the CLASSPATH now.

A JDBC driver will be used for connecting to Oracle. At the time of writing this guide, Oracle is providing a JDBC driver in a file called "classes111.zip". This file include the package "oracle.jdbc.driver" that implements the JDBC driver. You need to include this ZIP file as part of your CLASSPATH.

**Edit the "Registry.properties" File**

The "Registry.properties" file is a text file. It is easy to edit it directly using a text editor such as Notepad. The following lines are the entry for our HelpDesk data source. Each line will be explained below. These lines can be inserted at the end of the file.

```
HelpDesk.desc=Technical Support HelpDesk
HelpDesk.class=com.sqribe.jdbcacc.JDBCDataSource
HelpDesk.lib=oracle.jdbc.driver.OracleDriver
HelpDesk.load=
HelpDesk.conn=jdbc:oracle:oci7:@TechSupport.World
```

**Specify the Name of the Data Source**

As the example above indicates, the name of the data source, HelpDesk, is used as part of the property name. There are five properties that describe the HelpDesk data source:

| Property | Purpose |
|---|---|
| HelpDesk.desc | A description of this data source. |
| HelpDesk.class | The class name of the DDO driver. This is the name of the Java class that implements the DataSource interface. |
| HelpDesk.lib | A list of Java classes that need to be loaded as part of the initialization of this data source. |

| | |
|---|---|
| HelpDesk.load | A list of native libraries that need to be loaded as part of the initialization of this data source. |
| HelpDesk.conn | A connection string that the DDO driver understand. In the case of the JDBC Access driver, this is a JDBC URL. |

**Specify the DDO Driver**

A DDO driver is a Java class the implements the DataSource interface. For the JDBC Access driver, this class is "com.sqribe.jdbcacc.JDBCDataSource". This class in packaged in the "jdbcacc.jar" file. By specifying the name of the class as the value of the "HelpDesk.class" property we tell the DriverManager how to start this DDO driver.

**Specify Loading of the JDBC Driver**

The DDO JDBC Access driver can use any JDBC driver to access a relational data source. The relational data source is specified using a URL. The Java JDBC driver manager will use the URL to locate a suitable JDBC driver from among the drivers that are currently loaded into memory (that were loaded by the JVM class loader).

To ensure that a suitable driver can be found, one must often explicitly load the JDBC driver into memory. DDO supports that by allowing one to specify a list of Java classes to load in the "HelpDesk.lib" property. In this example, we specify the name of the Oracle JDBC driver's class, "oracle.jdbc.driver.OracleDriver". By loading this driver explicitly we make sure that the JDBC driver manager will successfully resolve the JDBC URL for the HelpDesk data source.

**Specify the JDBC URL**

The "HelpDesk.conn" specifies the data-source specific connection string. In the case of the DDO JDBC Access driver, this is a JDBC URL. The URL always start with "jdbc:". The next part selects the Oracle driver. The rest of the URL identifies the specific Oracle database and the connectivity method. For more information about constructing the URL one must refer to the JDBC driver documentation provided by Sun Microsystems, Inc.

Another example is using the Sun JDBC-ODBC bridge. The Registry entry below configures the HelpDesk data source to use the Sun JDBC-ODBC bridge. This

is assuming that one already has an ODBC data source configured under the name "HelpDesk".

```
HelpDesk.desc=Technical Support HelpDesk
HelpDesk.class=com.sqribe.jdbcacc.JDBCDataSource
HelpDesk.lib=sun.jdbc.odbc.JdbcOdbcDriver
HelpDesk.load=
HelpDesk.conn=jdbc:odbc:HelpDesk
```

Note that the "HelpDesk.class" property did not change. It is still the DDO JDBC Access driver. The "HelpDesk.conn" URL changed to specify "odbc" with the DSN (data source name) of "HelpDesk" (the name does not have to be "HelpDesk". It can be any name that was given to this ODBC data source). The "HelpDesk.lib" loads the Sun JDBC-ODBC bridge driver.

**Creating a New Data Source at Run-Time**

An application can define a new data source on the fly by using the "add()" method of the Registry class. Remember that the DataSourceManager manages a Registry of data sources. That Registry is an object of type Registry. A reference to this object can be obtained by using the "getRegistry()" method of the DataSourceManager class.

The example code below defines the HelpDesk data source on the fly.

```
// Create a data source on the fly
DataSourceManager.getRegistry().add(
    "HelpDesk",                                  // name
    " Technical Support HelpDesk ",              // desc
    "com.sqribe.jdbcacc.JDBCDataSource",         // class
    "oracle.jdbc.driver.OracleDriver",           // lib
    "",                                          // load
    " jdbc:oracle:oci7:@TechSupport.World ");    // conn
```

After successfully registering the "HelpDesk" data source, the application can obtain the DataSource interface by calling "DataSourceManager.getDataSource("HelpDesk")".

Note: If one loads a data source on the fly using the "add()" method of the Registry class it may prevent the automatic load of "Registry.properties". This is because "Registry.properties" is only loaded if the registry is empty. If it is desired that "Registry.properties" be loaded, one of two things can be done. One can explicitly load it with "DataSourceManager.getDataSourcesAdmin().load("Registry")", or you can have it automatically load before you add the new data source by calling the "getDataSource()"

or "getDataSources()" method of the DataSourceManager class. Calling these methods before adding the new data source—while the registry is empty—will cause "Registry.properties" to be loaded.

### *Establishing A Connection*

Once your application located a data source and obtained a DataSource interface, it can proceed to establish a connection to the data source. A connection is created using the "open()" call. This call returns a Connection interface.

Before you can call "open()" you must set the logon properties that are required to connect to this data source. Typically, you will need to set the "user" and "password" properties. However, DDO does not assume that this is always the case. DDO allows the data source to specify the logon properties and allows your application to discover these properties at run-time.

The code example below shows how the "user" and "password" properties are set and how the "open()" call is used to establish a connection.

```
import com.sqribe.access.*;
...
try {
    DataSource ds = DataSourceManager.getDataSource("accounting");
    if (ds == null) ...
    ds.getPropertySheet().setProperty("user", "scott");
    ds.getPropertySheet().setProperty("password", "tiger");
    Connection c = ds.open();
} catch (Exception e) {
    e.printStackTrace();
}
```

After we've located the data source and obtained a DataSource interface, we proceed to set the "user" and "password" properties. We obtain the DataSource PropertySheet and using the "setProperty()" method of the PropertySheet class we set the "user" and "password" properties. Note that these calls would fail with a PropertyException if the data source does not recognize a "user" or "password" property. Your application must be ready to handle this exception. The PropertySheet and PropertyException classes are defined in the comutil package ("com.sqribe.comutil"). This package contains the general-purpose classes that are used by DDO.

Once the properties that are required for logon have been set, a connection is easily established with an "open()" call. The "open()" call does not take any arguments. It returns a Connection interface. Your application will use this interface to access the data source.

If the "open()" call fails, a DataAccessException is thrown.

Your application cannot assume that "user" and "password" are the logon properties for any arbitrary data source. In the next section we will examine the general case in which the application checks for the logon properties for the given data source.

**Processing Logon Properties**

A DDO data source can be any application object that holds data—not necessarily a traditional DBMS. Therefore, we cannot assume that logon can always be established using a user name and password. That would be too limited of an assumption. In fact, a data source may require any number of items to establish a connection. These items can be anything that identifies the data to be accessed and the identity of the user who is accessing. This may include an account number, folder name, identification code, certificate and digital signature, to name a few examples.

An interactive application can obtain the list of logon properties from a data source at run-time and prompt the user for these properties. For each property, the application can obtain a description of the property, type, valid values, whether the property is required and whether it is secure. Properties can be grouped. For example, the "logon" property groups all logon properties.

A batch application can be programmed to pass properties to a data source based on prior knowledge of the specific data source. An example of that was shown earlier with the "user" and "password" properties.

The PropertySheet class provides access to PropertyDescription objects that describe its valid properties. PropertyDescription objects can be nested to represent grouped or nested properties.

Consider the following code example,

```
import com.sqribe.access.*;
import com.sqribe.comutil.*;

public class Example {
   public static void main(String[] args) {
      try {
❶        DataSource ds = DataSourceManager.getDataSource("accntng");
❷        PropertySheet prop = ds.getPropertySheet();
❸        list(prop.getPropertyDescription("logon"));
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   static void list(PropertyDescription pd) {
❹     PropertyDescription l[] = pd.getIndexes();
      if (l != null)
❺        for (int x = 0; x < l.length; x++) list(l[x]);
      else
❻        System.out.println(pd.getName());
   }
}
```

In step ❶, we locate the accntng data source and obtain a DataSource interface to it. In step ❷, we obtain the DataSource PropertySheet object. In step ❸, we call the "getPropertyDescription()" method of PropertySheet to obtain a PropertyDescription object for "logon". Every data source has a "logon" PropertyDescription that groups the logon properties. We then call the "list()" method of our Example class to list the PropertyDescription. Remember that a PropertyDescription can be nested to group additional PropertyDescription objects.

The "list()" method checks to see if there are any nested PropertyDescription objects by calling "getIndexes()" in step ❹, and then checking to see if it's null. If it is null, it means that there are no nested PropertyDescription objects for this PropertyDescription. If there are nested objects, we call "list()" recursively for each nested object in step ❺. Otherwise, we simply print the property name in step ❻. By traversing recursively we eventually reach all the properties that are nested under "logon".

The application can prompt the user for a value for each property. To do so, the application obtains additional attributes from the PropertyDescription objects. These attributes provide all the necessary information for the application to obtain logon attribute from the user and pass them to DDO. These attributes are summarized in the table below.

| Attribute | Purpose |
|---|---|
| Label | The label for prompting the user for this property. |

| | |
|---|---|
| Description | A description of the property that can be offered to the user as a help message. |
| Required | Specifies whether the user must supply a value for this property. |
| Secured | Specifies whether the value of this property is secured. This is for properties such as password that should not be echoed on the screen and should be encrypted. DDO performs the encryption automatically. |
| Validator | This is a name of a class that implements the PropertyValidator interface. Your application does not need to worry about this property since the Validator is typically supplied by either DDO or the DataSource driver. |
| ValidationType | Specifies the type of validation for the value of the property: This is a numeric value, 0 means no validation, 1 means that the value must fall inside a range, and 2 means that the value must be picked from a list. The validation is performed by the validator class specified in the Validator attribute. |
| ValidationValues | This is a list. If the ValidationType is range then the first item on the list is the minimum value and the second is the maximum value. Otherwise the items represent the valid choices for the value for this property. Your application can use these values to populate a list box. |

### *Discovering Capabilities*

Capabilities are read-only properties that describe capabilities and attributes of the data source. Like properties they can be grouped and nested. Like properties they are access via the PropertySheet class and are described using the PropertyDescription class. Their main use is to describe the data-access interfaces that the data source supports.

If the application is familiar with the data source, it probably already knows its capabilities. However, if one is building an ad-hoc application that can connect to various data sources, it is important to be able to discover the capabilities of the data source at runtime. To support a great variety of data sources and to ease the chore of writing a DDO driver, the DDO specification defines a minimal mandatory interface. Beyond the minimum, the driver is free to decide which interfaces to support—as long as it declares its capabilities.

The following are some examples of such capabilities. DDO allows the application to check if a data source supports each one of these capabilities.

| Capability | Description |
|---|---|
| command.supported | Indicates that the data source supports the execution of commands. A data source is not required to support any command language. The only required method is the "getData()" method that allows you to name an information object and obtain its data. |
| call.supported | Indicates that the data source supports the execution of procedures. A data source is not required to support procedures. |
| selector.supported | The only method of data retrieval that the data source must support is the "getData()" method. Even with the "getData()" method, the data source is only required to support naming an information object. The data source has the option of supporting the "getData(Selector)" method that allows your application to pass a Selector object to the "getData()" method in order to qualify the data retrieval. |
| transaction.supported | Indicates that the data source supports the Transaction interface. If it does not support the Transaction interface then a commit and rollback calls will be silently ignored. |
| md.supported | Indicates that the data source is multidimensional. You can pass a MDSelector object on the "getData(Selector)" call. MDSelector is a special kind of selector for multidimensional data sources. You can also obtain hierarchical dimension metadata by fetching child members using the "getChildren()" method of SchemaObjectColumn. This is useful for multidimensional databases. If the data source does not support this capability then a call to "getChildren()" on a SchemaObjectColumn would return null. The data source also supports the MDSchemaObject interface. |
| concurrent.connection .supported | Indicates whether or not the concurrent use of a connection is supported by the data source. When supported, multiple calls may be active through the same connection. The kind of concurrency supported is delineated by the specifications of the call, execute, selector, and transaction concurrency settings.<br>A call is said to be active if any of the Rowset objects that it returns are still active (still hold results pending). |
| JDBC.Database.get-IdentifierQuoteString | This is an example of a driver-specific capability. If your application is talking to the JDBC Access driver, it can obtain JDBC-specific capabilities.<br>The JDBC.Database.getIdentifierQuoteString capability tells your application what character to use for identifiers in SQL such as a column name that contains a space. Typically this is either single quote or double quote. |

### Checking for a Capability

The following code example checks to see if the data source supports the Transaction interface.

```
boolean checkTransactionSupport(DataSource ds) {
    PropertySheet prop = ds.getPropertySheet();
    Boolean supported =
        (Boolean)prop.getCapability("transaction.supported");
    if (supported == null) return false;  // capability is not defined
    return supported.booleanValue();
}
```

The following example checks for the "JDBC.Database.getIdentifierQuoteString" capability and uses it to compose a SQL statement.

```
try {
    // connect to the HelpDesk database
    DataSource ds = DataSourceManager.getDataSource("HelpDesk");
    PropertySheet prop = ds.getPropertySheet();
    prop.setProperty("user", "scott");
    prop.setProperty("password", "tiger");
    Connection c = ds.open();
    // check for the JDBC.Database.getIdentifierQuoteString
    prop = c.getPropertySheet();
    String quote = (String)prop.getCapability(
                        "JDBC.Database.getIdentifierQuoteString");
    // use the quote string to construct the SQL
    String sql = "select " + quote + "Employee Name" +
                 quote + " from " + quote + "Employees" + quote;
    // Based on the data source, the following should display either
    // select 'Employee Name' from 'Employees'
    // or
    // select "Employee Name" from "Employees"
    System.out.println(sql);
    c.close();
} catch ...
```

### *Obtaining Metadata*

Using the DDO API you can obtain rich metadata about the data objects that are accessible at the data source. DDO is designed such that you don't need any prior knowledge of the data objects. The metadata provides a complete description of how to access these objects, their parameters, and the data they provide.

To provide uniform metadata for very different data sources, DDO uses the following abstraction:

- A data source has one or more schemas. A schema is a grouping of data objects, procedures, or additional schemas.

  Note that some databases organize schemas within catalogs. In DDO these will be reflected as schemas within schemas. Also note that some data sources organize data objects and procedures in a hierarchy that can have any number of levels. DDO supports that with its recursive notion of schemas.
- DDO imposes no limit on the level of nesting of schemas within schemas.
- A data object is a generalized abstraction for any object that can provide its data on request. This includes tables, views, files, and business objects.
- A data object has a set of columns.
- A procedure is a generalized abstraction for any callable procedure or method that can be executed at the data source. This includes stored procedures and methods on business objects.
- A procedure may have parameters, a return value, and zero or more result sets.
- A procedure's result set has a set of columns.
- A column of a data object or a procedure's result set could represent a scalar item such as a date field, or a complex item such as a structure. The same is true for procedure parameters and return value.
- A column, parameter, or return value can have children that are themselves columns. This represents a hierarchy.
- A column hierarchy could represent a "dimension" in multidimensional (OLAP) terminology. It means that the column represents a hierarchy of members (for example departments in a hierarchical organization). By recursively enumerating the children of the column you can traverse the outline of a dimension.
- A column hierarchy could represent a structure or a table. In these cases by enumerating the children of this column you can list the fields of the structure or table. In the most general case this could also be a recursive process (nested structures and tables).

This is fairly abstract, but it will become clearer as we discuss the API and give example of its usage.

### *Obtaining Schema Information*

Using the Connection interface you can list the objects that are available at the data source. DDO groups these objects into schemas. A data source will always have at least one schema that groups the data objects and procedures that it provides. In the most general case, multiple schemas may be organized as a hierarchy. Each schema in the hierarchy may hold additional schemas as well as data objects and procedures.

To get at the top of that hierarchy, use the "getSchemas()" method of the Connection interface. This methods returns a "Schemas" objects. This object holds a list of objects that are contained within the current schema. As indicated earlier, the list may contain data objects, procedures, or additional schemas.

The following code example is a function that takes a Connection interface as a parameter and lists all the objects starting with the top of the hierarchy and recursively listing the schemas.

```
static void listConnection(Connection c) throws DataAccessException
{
❶    list(c.getSchemas(), 0);
}

static void list(Schemas schemas, int level)
{
    Enumeration enum;
    enum = schemas.elements();
    while (enum.hasMoreElements()) {
❷        Schema schema = (Schema)enum.nextElement();
        if (level > 0) indent(level);
❸        if (schema instanceof SchemaObject)
            System.out.println("data object: " + schema.getName());
❹        else if (schema instanceof SchemaProcedure)
            System.out.println("procedure: " + schema.getName());
❺        else {
            System.out.println("schema: " + schema.getName());
❻             Schemas children = schema.getChildren();
             if (children != null)
                 list(children, level + 1);
        }
    }
}

static void indent(int level)
{
    while (level-- > 0) System.out.print("    ");
}
```

To recurse through the schema hierarchy we use the recursive function "list()". This function takes an object of type "Schemas" as an argument. The "Schemas" class is a

collection of schema elements—objects of type "Schema"—data objects, procedures, and schemas. The other argument of the "list()" function is the level of recursion. It is used for indentation when the objects are listed.

In step ❶ we start at the top of the hierarchy by calling "getSchemas()" on the Connection interface. "getSchemas()" returns a "Schemas" object that holds all the top-level schemas and objects for this data source. We then call "list()" to list the schema recursively.

The "list()" function enumerates the items in the Schemas object. These items are of type Schema. See step ❷. Note that all the items in the schema are of type Schema. This includes data objects and procedures that are abstracted in the SchemaObject and SchemaProcedure classes, respectively. The "SchemaObject" and "SchemaProcedure" classes are subclasses of the Schema class.

In step ❸ we check the Schema object to see if it is a data object by checking if it is an instance of the "SchemaObject" class. If the object is of type SchemaObject, then we list it as a data object.

In step ❹ we check the Schema object to see if it is a procedure by checking if it is an instance of the "SchemaProcedure" class. If the object is of type SchemaProcedure, then we list it as a procedure.

When we get to step ❺, we've already accounted for the cases of data objects or procedures. The Schema object must therefore represent a schema. We list it as a schema and further list its contents by making a recursive call to "list()" in ❻.

The example above demonstrated the most general case in which we had no knowledge of the data source and had to traverse the schemas to discover all the objects. In the following examples we will examine some special cases.

**Listing Data Objects in a Schema**

In the following example, we assume that we are connecting to an Oracle database and want to list the tables and views that are accessible under user SCOTT. Here is the code:

```
try {
    DataSource ds = DataSourceManager.getDataSource("HelpDesk");
    PropertySheet prop = ds.getPropertySheet();
    prop.setProperty("user", "scott");
    prop.setProperty("password", "tiger");
    Connection c = ds.open();
    Schemas schemas = c.getSchemaObjects(c.getSchema("SCOTT"));
    Enumeration enum;
    enum = schemas.elements();
    while (enum.hasMoreElements()) {
        SchemaObject obj = (SchemaObject)enum.nextElement();
        System.out.println(obj.getType() + ": " + obj.getName());
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

One creates a Connection to the HelpDesk database. We then make a call to the "getSchemaObjects()" method of the Connection interface to get a list of the objects in the schema named "SCOTT". We then enumerate the elements in the schema. Each element is a SchemaObject. We print its type and its name. The type is a database specific name that represents the type of object, for example "TABLE" or "VIEW".

In a similar manner, one can list the stored procedure under the schema named "SCOTT" by making the following call:

```
Schemas schemas = c.getSchemaProcedures(c.getSchema("SCOTT"));
```

### *Obtaining Information about Data Objects*

Because DDO must deal with very different data sources it must provide a generalized abstraction of what a data source holds. As indicated in the previous section, a data source holds schemas that contain data objects and procedures.

In this section we will focus on these data objects. First, a few examples of data objects are:

- Tables and Views in a relational database. Basically any object one can "select" data from.
- A "cube" in a multidimensional database.
- A data file in a data source that provides access to files.
- An XML document.
- An object representing a collection of objects in an object database or application. For example, a list of employee objects.

DDO provides methods that allow you to list data objects and obtain their metadata. Once the object is located, a "SchemaObject" reference allows you to obtain the metadata for this object.

Here are some answers to the following type questions:: What can be done with such an object? How does one retrieve data from it? Can it be updated?:

At the very minimum, one can obtain data from this data object using the "getData()" method of the Connection interface. This method is most useful for data sources that don't support SQL or any other query language.

If the data source supports a query language, one can send queries that name this object to retrieve data from it.

Lastly, if the data source supports a command language such as SQL, one can also send DML (data manipulation) commands to it. For example, one can send an UPDATE statement to a data source that supports SQL.

Given a SchemaObject, one can look at some of its attributes:

| Attribute | Description |
|---|---|
| Name | The object name. |
| Description | A description or comment associated with this object. |
| Type | The object type. This is a data source specific string such as "TABLE", or "VIEW". |
| Parent | The schema that holds this object (if available). |

One can also obtain information about the columns of this data object. To do that, you can call the "getSchemaObjectColumns()" method of the Connection interface. This method takes a reference to a SchemaObject and returns a SchemaObjectColumns reference. One can also get column metadata by calling the "getMetaData()" method of SchemaObject.

Consider an example.

```
void listColumns(SchemaObject obj) throws DataAccessException {
    SchemaObjectColumns cols = obj.getMetaData().getSchemaObjectColumns();
    for (int i = 0; i < cols.size(); i++) {
        SchemaObjectColumn col = (SchemaObjectColumn)cols.elementAt(i);
        System.out.println("    " + col.getName() +
                           ", " + col.getDBTypeName());
    }
}
```

The function "listColumns" in our example takes a SchemaObject as an argument and lists its columns. We start by obtaining a SchemaObjectColumns reference for this object. This is done using the "obj.getMetaData().getSchemaObjectColumns()" call. The SchemaObjectColumns object is a vector of SchemaObjectColumn objects. The "count()" method returns the number of columns in this object and the "elementAt()" method returns each column. Note that we must cast the result of the "elementAt()" call to SchemaObjectColumn. For each column (SchemaObjectColumn) we print the name and database datatype name.

More about columns in described below in the section "Columns". The following reviews the metadata that is available for procedures.

### *Obtaining Information about Procedures*

Procedures are very powerful objects in DDO. They provide an abstraction for information objects that are parameterized.

The following are examples of procedures.

- Stored database procedures and functions in a relational database.
- Remote procedure calls (RPCs).
- Method invocation on objects in object databases and applications.
- Method invocation of COM or CORBA interfaces to business objects.

One can pass input parameters and receive values back on output parameters. The same parameter can be used for both input and output (INOUT parameter). Procedures can return a value. This is similar to a function call. Most of all, procedures can return data in multiple sets and DDO allows one to obtain descriptions for the columns of each such result set.

DDO provides robust support for procedure calls by allowing parameters to be complex structures. Similarly, result sets are not limited to flat tables and can hold complex structures.

Given a SchemaProcedure, one can look at some of its attributes.

| Attribute | Description |
|---|---|
| Name | The procedure name. |
| Description | A description or comment associated with this object. |
| Type | The object type. This is a data source specific string such as "Procedure", or "Function". |
| Parent | The schema that holds this object (if available). |

To get the metadata describing parameters, return value, and columns, one can call the "getProcedureMetaData()" method of the Connection interface. This method takes a reference to a SchemaProcedure and returns a ProcedureMetaData reference. One can also get metadata by calling "getMetaData().getProcedureMetaData()" on SchemaProcedure.

**Listing Procedure Parameters**

The example below demonstrates listing the parameters of a procedure and displaying for each parameter its kind—input, output, or both.

```
void listParameters(ProcedureMetaData meta) {
   SchemaProcedureColumns params = meta.getParameters();
   if (params == null)
      System.out.println("  No parameters for this procedure.");
   else
      for (int i = 0; i < params.size(); i++) {
         SchemaProcedureColumn col = meta.getParameter(i);
         String kind = null;
         switch (col.getUse()) {
            case SchemaProcedureColumn.PARMIN:
               kind = "IN";
               break;
            case SchemaProcedureColumn.PARMOUT:
               kind = "OUT";
               break;
            case SchemaProcedureColumn.PARMINOUT:
               kind = "INOUT";
               break;
         }
         System.out.println("  parameter: " + col.getName()
                            +  " - " + kind);
      }
}
```

The "listParameters()" method takes a ProcedureMetaData object as an argument. By calling the "getParameters()" on this object we obtain a SchemaProcedureColumns object representing the parameters list of this procedure. If "getParameters()" returns null it means that this procedure has no parameters. Otherwise, we list each parameter. The

"getUse()" method of SchemaProcedureColumn returns the kind of parameter—input, output, or both.

### Determining the Return Value

The example below demonstrates how to check for the procedures return value.

```
void listReturnValue(ProcedureMetaData meta) {
   SchemaProcedureColumn retValue = meta.getReturnValue();
   if (retValue == null)
      System.out.println("  No return value for this procedure.");
   else
      System.out.println("  return Value: " + retValue.getName()
                         +  " - " + retValue.getDBTypeName());
}
```

The "listReturnValue()" method takes a ProcedureMetaData object as an argument. By calling the "getReturnValue()" on this object we obtain a SchemaProcedureColumn object representing the return value of this procedure. If "getReturnValue()" returns null it means that this procedure has no return value. Otherwise, display the return value along with its data source specific type name.

### Listing Procedure Result Sets

The example below demonstrates listing the result sets of a procedure and listing the columns of each result set.

```
void listResultSets(ProcedureMetaData meta) {
   Vector resultSets = meta.getResultSets();
   if (resultSets == null)
      System.out.println("  No result sets for this procedure.");
   else
      for (int i = 0; i < resultSets.size(); i++) {
         System.out.println("  Result set " + i + ":");
         listResultSet(meta, i);
      }
}
void listResultSet(ProcedureMetaData meta, int i) {
   int columnCount = meta.getResultSet(i).size();
   for (int j = 0; j < columnCount; j++) {
      SchemaProcedureColumn col = meta.getResultSetColumn(i,j);
      System.out.println("    column: " + col.getName()
                         +  " - " + col.getDBTypeName());
   }
}
```

The "listResultSets()" method takes a ProcedureMetaData object as an argument. By calling the "getResultSets()" on this object we obtain a Vector (java.util.Vector) object representing an array of one or more result sets. If "getResultSets()" returns null it

means that this procedure has no result sets. Otherwise, we call "listResultSet()" to list the columns for each result set.

The "listResultSet()" method takes a ProcedureMetaData object and a result set number as arguments. By calling the "getResultSet(i).size()" we obtain the number of columns in this result set. We then fetch metadata for each column using the "getResultSetColumn(i, j)" call. In this call, "i" represents the result set number and "j" represents the column within that result set.

### *Columns*

Columns describe data items in data objects and in procedure result sets. Columns also describe procedure parameters and return value. In previous sections we saw how to obtain columns for data objects and procedures. These columns are abstracted in the SchemaObjectColumn and ProcedureObjectColumn classes. SchemaProcedureColumn is a subclass of SchemaObjectColumns. It shares all the attributes of SchemaObjectColumns. It also provides additional attributes.

The following table lists the attributes that are common to all columns.

<table>
<tr><th>Attribute</th><th colspan="2">Description</th></tr>
<tr><td>Name</td><td colspan="2">The name of the column.</td></tr>
<tr><td>Size</td><td colspan="2">The width of the field.</td></tr>
<tr><td>Precision</td><td colspan="2">The precision for numeric columns.</td></tr>
<tr><td>Scale</td><td colspan="2">The scale for numeric columns.</td></tr>
<tr><td>DBType</td><td colspan="2">A data source specific number representing the data source specific datatype.</td></tr>
<tr><td>DBTypeName</td><td colspan="2">A data source specific datatype name.</td></tr>
<tr><td rowspan="10">FieldType</td><td colspan="2">This is the DDO datatype for this column. When data is retrieved from the data source, this column will return a field of this type. The FieldType is an integer number that is one of the constants that are defined in the Field class.</td></tr>
<tr><td>Constant</td><td>Description</td></tr>
<tr><td>Field.Text</td><td>Text string field.</td></tr>
<tr><td>Field.Number</td><td>Numeric field. This can be an integer, double, or Decimal.</td></tr>
<tr><td>Field.Date</td><td>A date or date and time field.</td></tr>
<tr><td>Field.Boolean</td><td>A true or false value.</td></tr>
<tr><td>Field.Binary</td><td>Binary raw data (stream of bytes).</td></tr>
<tr><td>Field.Row</td><td>A structure.</td></tr>
<tr><td>Field.Rowset</td><td>A table.</td></tr>
<tr><td>Field.Object</td><td>An arbitrary object.</td></tr>
</table>

SchemaProcedureColumn describes procedure parameters and return value. It has an additional attribute.

| Attribute | Description | |
|---|---|---|
| Use | The use of this column. This is an integer number that is one of the constants that are defined in the SchemaProcedureColumn class. | |
| | **Constant** | **Description** |
| | PARMIN | An input parameter. |
| | PARMINOUT | A parameter used for both input and output. |
| | PARMOUT | An output parameter. |
| | RESULTCOLUMN | A column in a result set. |
| | RETURNVALUE | A procedure's return value. |
| | UNDEFINED | Undefined. |

### *Requesting Data*

There are three fundamental ways of requesting data in DDO: getData, execute, and call.

#### getData

The most basic method of getting data in DDO is using the "getData()" method. This method simply names an object and requests its data. This method can be supported by the simplest of data sources, those that do not support any query language or procedure-call mechanism.

#### execute

The second method of requesting data in DDO is executing a command such as a SQL SELECT statement. The command is a text string that is passed through to the data source driver. The command may be parameterized using "?" in the command text.

#### call

The third method of requesting data in DDO is calling a procedure. DDO allows your application to call procedures directly (no need to construct a data-source

specific statement for that purpose). You can pass parameters when calling a procedure and obtain output parameters, return values, and multiple result data sets.

In the following section "Selecting and Filtering" we will examine the "getData()" method and the associated Selector class. Using the selector class one can specify the columns that should be returned.

### *Retrieving Data with getData*

The most basic method of data retrieval in DDO is the "getData()" method. The method takes the name of a data object as an argument. One can pass the name as a string or a SchemaObject reference. The "getData()" method retrieves all the data for the named object. Results are returned as a Rowset.

Consider an example.

```
try {
    DataSource ds = DataSourceManager.getDataSource("HelpDesk");
    PropertySheet prop = ds.getPropertySheet();
    prop.setProperty("user", "scott");
    prop.setProperty("password", "tiger");
    Connection c = ds.open();
    Rowset rowset =
        c.getData(c.getSchemaObject(new String[] { "SCOTT", "EMP"}));
    // print headings
    for (int i = 0; i < rowset.getFieldCount(); i++)
        System.out.print(rowset.getField(i).getName()+ ",");
    System.out.println("");
    while (rowset.next()) {
        // print data
        for (int i = 0; i < rowset.getFieldCount(); i++)
            System.out.print(rowset.getField(i) + ",");
        System.out.println("");
    }
    c.close(); // close the connection to the data source
} catch (Exception e) {
    e.printStackTrace();
}
```

In the example, we use the "getData()" method with a reference to the object. We get a reference to this object using the "getSchemaObject()" method of Connection. We pass "SCOTT" followed by "EMP" as the path to the object. The interpretation of this path is specific to the data source. Assuming that the data source has a flat set of schemas containing tables, the path would be a schema "SCOTT" with a table "EMP". "getData()" returns a Rowset interface with all the data in the EMP table. This is equivalent to executing the command "SELECT * FROM SCOTT.EMP".

### *Processing Results using the Rowset Interface*

Rowset is an interface. This means that DDO does not prescribe the implementation of a Rowset. This gives the data source driver great flexibility in handling data while providing the application a consistent interface.

In particular, the driver is free to implement just in time retrieval. This means that the Rowset does not actually hold all the data. The driver may fetch a record when the application calls the "next()" method. The benefit of this behavior is that the driver does not need to hold the entire result set in memory and can handle very large result sets. This process was referred to earlier as "streamed result sets."

The Rowset is self-describing. Using the "getFieldCount()" method one can determine the number of fields in the Rowset. Each field is an instance of the abstract Field class. A Field object describes its type, size, and structure. In particular, a Field may be a complex structure such as a Row or Rowset.

The benefit of Rowset being self-describing is that the application does not need to hardcode the expected type and sizes of each field. It can discover these attributes at run-time. The following example demonstrates how the datatype of the fields can be determined while processing a Rowset.

```
static void printRowset(Rowset rs) throws DataAccessException {
   // write results in an HTML table
   System.out.println("<table><tr>");
   int fieldCount = rs.getFieldCount();
   // DateFormat object for printing dates
   DateFormat df = DateFormat.getDateInstance(DateFormat.LONG);
   // print headings
   for (int i = 0; i < fieldCount; i++)
       System.out.println("<td>" + rs.getField(i).getName() + "</td>");
   // print the data row by row
   while (rs.next()) {
      System.out.println("</tr><tr>");
      for (int i = 0; i < fieldCount; i++) {
         Field f = rs.getField(i);
         if (f.isNull()) {
            System.out.println("<td> </td>");
            continue;
         }
```

```
            switch (f.getType()) {
               case Field.Boolean:
               case Field.Text:
                  System.out.println("<td align=left>" + f + "</td>");
                  break;
               case Field.Number:
                  if (f instanceof DecimalField &&
                        ((DecimalField)f).isCurrency())
                     System.out.println("<td align=right>$" + f + "</td>");
                  else
                     System.out.println("<td align=right>" + f + "</td>");
                  break;
               case Field.Date:
                  System.out.println("<td align=left>"
                        + df.format(((DateField)f).dateValue()) + "</td>");
                  break;
               case Field.Row:
               case Field.Rowset:
               case Field.Object:
               case Field.Binary:
                  System.out.println("<td align=center>n/a</td>");
                  break;
            }
         }
      }
      System.out.println("</tr></table>");
      rs.close(); // close the result set
}
```

The "printRowset()" function in our example takes a Rowset as an argument. It has no knowledge of how the Rowset was obtained. The Rowset may be the result of "getData()", or the result of executing a command such as a SQL statement, or it could be a result set from an execution of a stored procedure. In all cases, the Rowset is processed in the same manner.

The Rowset maintains a current row with a fixed number of fields. A call to the "next()" method of the Rowset will populate the current row with the next row of data. "next()" returns a boolean value of "true" for as long as records are available. Calling "next()" after the last row returns "false".

For each row, each field is processed. One starts by checking the value for null using the "isNull()" method. If the field is null we print an empty cell and proceed to the next field.

The field type is checked using the following statement.

```
            switch (f.getType()) {
```

The "getType()" method returns an integer that matches one of the constants that are defined in the Field class. Our example takes different action depending on the type of the

field although we could simply print all fields as strings using the "toString()" method of the Field class.

Field types are summarized in the following table.

| Type | Description |
|---|---|
| Boolean | A true/false value. The field is an instant of BooleanField. |
| Binary | An array/stream of bytes. The field is an instant of BinaryField. It may also be an instance of LongBinaryField. |
| Date | A java.util.Date value. The field is an instant of DateField. |
| Number | A numeric value. The field is an instant of a subclass of the abstract NumberField class. This means that the field is an instance of IntegerField, DoubleField, or DecimalField. |
| Object | An arbitrary Java Object. The field is an instant of ObjectField. |
| Row | A Row. The field is an instant of RowField. |
| Rowset | A Rowset. The field is an instant of RowsetField. |
| Text | A String. The field is an instant of TextField. |

### *Selecting and Filtering*

Using the "getData()" method to retrieve data from an object is very simple. To allow for a more selective retrieval without requiring the data source to support a full-blown command language, DDO introduces the concept of a Selector.

A Selector specifies the columns that the application wishes to retrieve (rather than unconditionally retrieving all the columns) as well as simple selection criteria. These are discussed in the section "Obtaining Hierarchical and Multidimensional Data"..

How to use a Selector to pick the desired columns.

```
DataSource ds = DataSourceManager.getDataSource("CSVFiles");
Connection c = ds.open();
Selector selector = new Selector();
selector.setObject(c.getSchemaObject("Employee.csv"));
selector.includeColumn("Name");
selector.includeColumn("Salary");
selector.includeColumn("HireDate");
Rowset rs = c.getData(selector);
printRowset(rs);
```

In this example we use the DDO CSV Access driver. This driver allows access to CSV (comma separated values) files. While this driver does not support SQL, it does support the selector interface.

The code example above demonstrates how we use the Selector class to specify the data that we want. One starts by instantiating an empty selector. We then set the Object attribute to the desired CSV file. We pass a SchemaObject to "setObject()". We obtain this object using the "getSchemaObject()" method of the Connection interface. Note that the Registry already contains information for this data source. The information includes a starting disk folder. This defines the default schema for this object. In other words, we are requesting data from an object named "Employee.csv" that resides in the default schema.

We further request the columns that should be included when we issue the "getData()" cal and pass this selector. Note that we must specify the object before we specify the columns so that the Selector is able lookup the columns. The result is a Rowset containing the three fields that we have requested.

The Selector class provides an SQL-like syntax for making the same request. This is demonstrated in the example code below. By passing the select statement on the Selector constructor we have defined both the desired object name and the desired columns. We also pass the Connection interface to allow the Selector to lookup objects for us. Note that we use double quotes around column or object names that may contain spaces or a dot.

```
Selector selector = new Selector(c,
                "select Name, Salary, HireDate from \"Employee.csv\"");
```

### *Executing Commands*

If the data source supports the command interface, then the application can send command statements to the data source for execution. This is most powerful for data sources that support rich command language such as SQL. Using a language one can specify complex queries that include data selection, aggregation, and composition. For example, if the database supports SQL one can perform joins and group and sort the results.

Remember that one can check that a data source supports the command interface by checking for that capability. See the section "Discovering Capabilities" for how to check for a specific capability.

A command can be parameterized and the application can supply values for these parameters at run time. This is demonstrated in the following example.

```
...
    Connection c = ds.open();
    Command command = new Command("select empno, ename, hiredate from " +
                      "emp where deptno = ?");
    command.setParameter(0, new IntegerField(20));
    Rowset rs = c.execute(command);
...
```

In the example we construct a Command object with a SQL statement. Note that the "?" symbol is used as a marker for a parameter in the statement. We supply a value for this parameter by passing a Field object to the "setParameter()" method of the Command class. Actually, the "setParameter()" method belongs to the ParameterList class from which Command is derived.

The "setParameter()" method takes two arguments. The first is the parameter number, zero being the first. The second argument is a Field with a value for this parameter. In our example we construct an IntegerField with a value of 20 for department 20. Using a Field object to supply the value is most useful when you bind the result of one command as a parameter to another command.

The command is executed using the "execute()" method of the Connection interface. The method returns a Rowset object. The Rowset is processed as described in the section "Processing Results using the Rowset Interface".

A command may return a single row or even a single value. DDO still returns a Rowset, however the Rowset may have a single row and a single field. For example, consider a SQL Update statement. The only information returned from an Update is the number of database rows that were effected by the update. Let's see the code:

```
...
    Connection c = ds.open();
    Command command = new Command("update emp set sal = sal * 1.1");
    Rowset rs = c.execute(command);
    rs.next();
    System.out.println(rs.getField("COUNT") + " records were updated.");
    c.close(); // close the connection to the data source
...
```

In this example we execute an Update SQL statement. This statement does not return data, but it does return a row count for the number of rows processed. The DDO JDBC Access driver will return the row count in a Rowset that has a single row and a single field. The field is named "COUNT". The call to "getField("COUNT")" retrieves that field and then we print it. This variant of "getField()" locates a field by name. A faster way to

get a field is by position number where zero is the first field. We could alternatively code our print statement as follows.

```
System.out.println(rs.getField(0) + " records were updated.");
```

**Note about Database Cursors**

The DDO JDBC Access driver uses database cursors in a way that is completely transparent to the application. Under the cover, the driver maintains a pool of cursors (JDBC PreparedStatement objects). When a command is executed, the driver first checks to see if a cursor already exists for this command. The cursor pool improves performance by avoiding repeated prepare operations of the same SQL statement.

DDO expects that drivers implement such performance optimizations under the cover and in a manner that is transparent to the application. This is important for delivering excellent performance without cluttering the API with data source specific methods such as cursor management methods.

***Calling Procedures and Processing Call Results***

To execute a procedure you use the "call()" method of the Connection interface. Your application will pass an argument that identifies the procedure and can optionally pass a parameter list. The parameter list holds values for the input parameters of the procedure. Each parameter in the list is an object of type Field. This is useful when you want to pass a field that you just retrieved from the data source as an input parameter to the procedure call.

If the value that you need to pass as a parameter is not a Field, you will need to construct a Field to hold that value. DDO provides several methods for constructing a field.

- Your application can use the "new" operator to construct a field such as BooleanField, BinaryField, or DateField. Many of these methods can construct a field and set its value at once.
- Your application may use the static methods of the ParameterFactory class to create Fields and supply them with the actual value.

Once you have constructed the parameter list you can perform the "call()" and obtain the results. Let's look at an example:

```
public static void main(String[] args) {
   try {
      DataSource ds = DataSourceManager.getDataSource("Sales");
      Connection c = ds.open();
      // prepare the parameter list
❶     ParameterList params = new ParameterList(new Field[] {
                new DateField("1/1/98"), new DateField("12/31/98") });
      // call the procedure
❷     SchemaProcedure proc = c.getSchemaProcedure("Sales by Year");
❸     CallResults results = c.call(proc, params);
      // process all rowsets
      Rowset rs;
❹     while ((rs = results.getOutputRowset()) != null)
         printRowset(rs);
      // check for return value
❺     Row retval = results.getReturnValue();
      if (retval != null) printRow(retval);
      // check for output parameters
❻     Row outparams = results.getOutputParams();
      if (outparams != null) printRow(outparams);
      // close the call results
❼     results.close();
      c.close(); // close the connection to the data source
   } catch (Exception e) {
      e.printStackTrace();
   }
}
static void printRow(Row row) throws DataAccessException {
   for (int i = 0; i < row.getFieldCount(); i++) {
      Field f = row.getField(i);
      System.out.println("  " + f.getName() + ": " + f);
   }
}
```

In step ❶, after we have successfully connected to the Sales data source, we construct the parameter list. In our example, we assume that the procedure we are about to call takes two date input parameters. We construct a ParameterList object by passing an array of fields to the constructor. The fields hold the actual values for the parameters for this procedure call.

In step ❷, we locate the procedure and obtain a SchemaProcedure object for it. This is done using the "getSchemaProcedure()" method of Connection. If a procedure with this name does not exist, the method will throw a DataAccessException.

In step ❸, we make the call and obtain a CallResults object. The CallResults allows your application to obtain all the data that is returned from the procedure. This includes multiple result sets, output parameters, and a return value. Note that you should process the result sets before obtaining the values for the output parameters and the return value.

In step ❹, we process the result sets. In general, a procedure may return any number of result sets. It may return no result sets, it may return a single result set, or it may return multiple result sets. In our example we make repeated calls to the "getOutputRowset()" method of the CallResults class until there are no more result sets.

In step ❺, we process the return value. CallResults returns this value as a Row. This is useful when the value being returned is a structure. If you know that your procedure will return a scalar value (a single field), then you can code step 5 as follows:

```
❺       Row retval = results.getReturnValue();
        if (retval != null)
            System.out.println("return value: " + retval.getField(0));
```

In step ❻, we process output parameters. CallResults returns output parameters as a row in which each field represents a single output parameter. The ordering of the fields corresponds to the order of the output parameters of the procedure.

In step ❼, we close the CallResults object. This signals the driver that the execution context of this procedure call can be released.

### *Performing Transactions*

Business Intelligence applications do more than read data. They often update request tables, log tables, status fields, and more. Moreover, an application may stage data into intermediary storage so that multiple passes can be easily performed and multiple reports can be generated.

DDO supports update activity to the database in several ways. A Command that is executed via the "execute()" method of Connection can perform any operation on the data source. In particular, it can create new objects and populate them with data.

DDO supports procedure calling via the "call()" method of Connection. DDO imposes no limit to what a procedure can do. The data source may allow the user to call procedures that update data and manipulate objects.

To group data changes into transactions, DDO provides the transaction interface. If the data source supports transactions, the call to the "getTransaction()" method of the Connection interface will return a Transaction interface. Using this interface, you application can start a transaction and complete a transaction with either a commit or a rollback.

Here is a simple example. In the example "c" is a Connection reference for a valid connection to our HelpDesk Oracle database.

```
try {
   c.getTransaction().beginTransaction();
   c.execute(
      new Command("update emp set sal = sal * 1.1 where job = 'CLERK'"));
   c.execute(
      new Command("update emp set sal = sal * 1.1 where job = 'MANAGER'"));
   c.getTransaction().commit();
   c.close();
   } catch (DataAccessException e) {
      try {
         if (c != null) c.getTransaction().rollback();
      } catch (Exception e2) {
         e2.printStackTrace();
      }
      e.printStackTrace();
}
```

We start by calling "beginTransaction()" on the transaction interface. This call is always required. We then execute two update statements. If an error occurs during the updates, then we catch and exception and rollback all the changes by calling the "rollback()" method of the Transaction interface. Otherwise, if the two updates are successful we call the "commit()" method. In this example we did not have to call "commit()" because the "close()" method on the connection will also commit any pending transaction.

### *Obtaining Hierarchical and Multidimensional Data*

DDO directly supports multidimensional databases (also called OLAP servers). These databases organize data to support multi-level aggregation and analysis. They define a data set—also called a hypercube—in terms of multiple dimensions. Each dimension represents a key aspect of the data. For example, in sales data dimensions typically include product, territory, organization units, and time. These represent what was sold, where it was sold, who sold it, and when. Sales data can therefore be analyzed along these dimensions.

Each dimension typically defines a hierarchical structure. This is key for data aggregation. For example, territory can define a hierarchy or geographical regions. At the top of the hierarchy, data is summarized for all regions. Going one level down, for example, can divide the world into North America, Europe, etc. North America can further be divided into countries and then states.

DDO maps multidimensional concepts to DDO objects as follows.

| Concept | DDO Objects |
|---|---|
| Hypercube | Data object (SchemaObject). |
| Dimension | Column (SchemaObjectColumn). |
| Dimension Hierarchy | A hierarchy of SchemaObjectColumn object. At the top of the hierarchy there is a SchemaObjectColumn object for each dimension. A "getChildren()" call on this column will return the first-generation members of that dimension hierarchy. A "getChildren()" call on a first-generation member will return second-generation members. You can continue down the hierarchy until "getChildren()" returns null.<br>You can also obtain information about levels and generations using the MDSchemaObject interface. |
| Measures | Numeric column (SchemaObjectColumn). |

Using DDO, your application can "walk" the dimension and discover all the members, generations, and levels.

Here is an example.

```
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.Enumeration;
public class test19 {
 public static void main(String[] args) {
    try {
       DataSource ds = DataSourceManager.getDataSource("Essbase");
       PropertySheet prop = ds.getPropertySheet();
       prop.setProperty("user", args[0]);
       prop.setProperty("password", args[1]);
       Connection c = ds.open();
       Enumeration enum = c.getAllSchemasObjects().elements();
       while (enum.hasMoreElements()) {
          SchemaObject cube = (SchemaObject)enum.nextElement();
          listCube(cube);
       }
       c.close();
    } catch (Exception e) {
       e.printStackTrace();
    }
 }
 static void listCube(SchemaObject cube) throws DataAccessException {
    System.out.println("----------------------");
    System.out.println(cube.getName());
    System.out.println("----------------------");
    SchemaObjectColumns cols = cube.getMetaData().getSchemaObjectColumns();
    listColumns(cols, 1);
 }
```

```
static void listColumns(SchemaObjectColumns cols, int level) {
   for (int i = 0; i < cols.size(); i++) {
      SchemaObjectColumn col = (SchemaObjectColumn)cols.elementAt(i);
      indent(level);
      printCol(col);
      SchemaObjectColumns children =
                              (SchemaObjectColumns)col.getChildren();
      if (children != null)
         listColumns(children, level + 1);
   }
}
static void printCol(SchemaObjectColumn col) {
   String name = col.getName();
   String desc = col.getDesc();
   if (desc != null && desc.length() > 0)
      System.out.println(name + " (" + desc + ")");
   else
      System.out.println(name);
}
static void indent(int level) {
   while (level-- > 0)
      System.out.print("    ");
}
}
```

The example starts off by obtaining a list of all the data objects in this data source using the "getAllSchemasObjects()" method of the Connection interface. We then go through the list and call "listCube()" for each data object. Remember that each data object in a multidimensional database represents a hypercube.

The "listCube()" method prints the name of the hypercube and then calls "listColumns()" to lists the columns of the hypercube. Each column represents a dimension except for the last column that represents the numeric data.

The "listColumns()" method is a recursive method that recurse through the hierarchy of members within a dimension. The child members of a dimension or a member are obtained with a "getChildren()" call. Note that in the case of a SchemaObjectColumn, "getChildren()" will always return SchemaObjectColumns.

The "printCol()" method prints the name of each column. This is the name of the dimension or member of the dimension. Note that some multidimensional databases use the name of the member as a unique identifier. The member can also have an alias that is more suitable for display in a report and can use a localized language. If such an alias is available, you will find it in the description attribute of the column (see "col.getDesc()"). If a description is available, "printCol()" will display it along with the column name.

How to get metadata for a multidimensional database was illustrated. How to retrieve the data will now be illustrated.

DDO applications do not have to be familiar with multidimensional concepts or even be aware that the data source is multidimensional. Therefore, DDO provides two modes for data retrieval:

- Using a regular Selector. The application is not "multidimensional aware".
- Using a multidimensional Selector (MDSelector). The application is "multidimensional aware".

**Retrieving Multidimensional Data Using a Regular Selector**

In the previous section it was shown that listing the objects in a multidimensional data is exactly the same as listing the objects in any data source. Moreover, listing the dimensions and measures of a hypercube object is identical to listing columns of a table object.

Therefore, we can retrieve data from a multidimensional data source by simply naming an object in a "getData()" call, or constructing a Selector object that names the hypercube object and includes selected dimensions using the "includeColumn()" method of Selector.

In both cases you will get the data as a rowset in which every dimension and measure is a field. The data is down to the lowest level (level 0) and there is a row for every intersection of the given dimensions (every cell in the hypercube). If you use a selector to pick specific dimensions, then the data is summarized across the other dimensions.

consider this example:

```
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.Enumeration;
public class test21 {
   public static void main(String[] args) {
      try {
         DataSource ds = DataSourceManager.getDataSource("Essbase");
         PropertySheet prop = ds.getPropertySheet();
         prop.setProperty("user", args[0]);
         prop.setProperty("password", args[1]);
         Connection c = ds.open();
         SchemaObject cube =
               c.getSchemaObject(new String [] { "Sample", "Basic" } );
❶        Selector selector = new Selector();
         selector.setObject(cube);
         selector.includeColumn("Year");
         Rowset rs = c.getData(selector);
         printRowset(rs);
         c.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   static void printRowset(Rowset rs) throws DataAccessException {
      int fieldCount = rs.getFieldCount();
      // print the data row by row
      for (int rowCount = 0; rs.next(); rowCount++) {
         StringBuffer line = new StringBuffer();
         for (int i = 0; i < fieldCount; i++) {
             line.append(rs.getField(i).toString());
             line.append('\t');
         }
         System.out.println(line);
      }
      rs.close();  // close the rowset
   }
}
```

In step ❶, we create a Selector object and set its object to our hypercube and pick one dimension called "Year".

And here is the output:

```
Jan     8024.0
Feb     8346.0
Mar     8333.0
Apr     8644.0
May     8929.0
Jun     9534.0
Jul     9878.0
Aug     9545.0
Sep     8489.0
Oct     8653.0
Nov     8367.0
Dec     8780.0
```

In our example, level 0 of the "Year" dimension is the month. Since this is the only dimension we've selected, the data is summarized along this dimension.

**Retrieving Multidimensional Data Using MDSelector**

When retrieving data from a multidimensional database, there are several things to specify:

- Dimensions and measures to include.
- Members to include. This is the multidimensional way of restricting the data to a subset. For example, to get the data from a single month, you can select the "March, 1999" member of the time dimension.
- Level of aggregation. For example, we may want the data summarized into weekly number, monthly numbers, or quarterly numbers. The weeks, months, and quarters are levels in the time dimension hierarchy.

DDO supports these kind of selections using a specialized kind of a Selector object called a MDSelector. The MDSelector class is a subclass of Selector. It builds on the Selector class's capability to select an object and columns. It adds the ability to select members and level of aggregation.

Consider an example.

```
SchemaObject cube =
        c.getSchemaObject(new String [] { "Sample", "Basic" } );
MDSelector selector = new MDSelector();
selector.setObject(cube);
selector.includeColumn("Year");
selector.includeColumn("Product");
selector.setColumnLevel(0, 1);
selector.setColumnGeneration(1, 2);
Rowset rs = c.getData(selector);
printRowset(rs);
```

In this example we use instantiate a MDSelector instead of a Selector. We "setObject()" to our hypercube ("Basic" in schema "Sample") and select the "Year" and "Product" dimensions. So far, this is no different than selector. Now, we choose the level of aggregation. The statement "setColumnLevel(0, 1)" sets the level for the first column (column 0—Year) to be level 1. This means one level higher than the most detailed level. If you think of the dimension hierarchy as a tree, then level 0 is the leaves of the tree and level 1 is their immediate parents. The statement "setColumnGeneration(1, 2)" sets the level for the second column (column 1—Product)

to be generation 1. Specifying the aggregation in terms of generation is useful, especially if the hierarchy tree is not balanced with some leaves deeper than others. Generation 1 is the top, generation 2 is the immediate children of the root of the tree.

And here is the output:

```
Qtr1    100     7048.0
Qtr1    200     6721.0
Qtr1    300     5929.0
Qtr1    400     5005.0
Qtr1    Diet    7017.0
Qtr2    100     7872.0
Qtr2    200     7030.0
Qtr2    300     6769.0
Qtr2    400     5436.0
Qtr2    Diet    7336.0
Qtr3    100     8511.0
Qtr3    200     7005.0
Qtr3    300     6698.0
Qtr3    400     5698.0
Qtr3    Diet    7532.0
Qtr4    100     7037.0
Qtr4    200     7198.0
Qtr4    300     6403.0
Qtr4    400     5162.0
Qtr4    Diet    6941.0
```

Now suppose that we only wanted to see the first two quarters. Moreover, we only want to see product groups 100, 200, 300, and 400 ("Diet" is a grouping of products that are already counted under 100, 200, 300, or 400). We want to restrict our selection to specified members. DDO allows that using the MDSelector "setColumnMembers()" method. Here is an example:

```
SchemaObject cube =
        c.getSchemaObject(new String [] { "Sample", "Basic" } );
MDSelector selector = new MDSelector();
selector.setObject(cube);
selector.includeColumn("Year");
selector.includeColumn("Product");
selector.setColumnLevel(0, 1);
selector.setColumnGeneration(1, 2);
selector.setColumnMembers(0,
          new String[] { "Qtr1", "Qtr2" });
selector.setColumnMembers(1,
          new String[] { "100", "200", "300", "400" });
Rowset rs = c.getData(selector);
printRowset(rs);
c.close();
```

We used the version of the "setColumnMembers()" method that takes member names as strings. This is assuming that member names are unique. Otherwise you must

obtain the outline metadata for the dimension and pass members to the "setColumnMembers()" method as SchemaObjectColumn objects.

And here is the output:

```
Qtr1    100     7048.0
Qtr1    200     6721.0
Qtr1    300     5929.0
Qtr1    400     5005.0
Qtr2    100     7872.0
Qtr2    200     7030.0
Qtr2    300     6769.0
Qtr2    400     5436.0
```

CREATING A DDO DRIVER

The current invention (DDO) makes the task of writing a driver for a new data source easy in the following ways:

- DDO only requires a driver to implement a minimal set of interfaces. Many of the interfaces are optional. In particular, a DDO data source is not required to support a query language or procedure calling, and it is not required to support transactions.
- A DDO driver declares its capabilities in a Capabilities file (**505 in Fig. 5**). No coding is required.
- DDO provides base classes that implement all the interfaces **507** . These classes provide more than just default behavior. They provide all the functionality of managing DDO metadata. They also provide much of the common code that all drivers would otherwise have to implement.
- DDO provides a framework for exceptions and localized error messages **503**. The messages are easily organized in resource files and can be managed using the tools that are included with the DDO SDK.

Referring now to **Figure 7** an exemplary process for creating a DDO driver is described. In **Fig. 7** the process begins by creating the Properties, Capabilities and Message files **703**, then the DataSource interface is implemented **705**. The Properties, Capabilities and Message files are refined based on the implementation of the Data Source Interface **706**. This step is followed by implementing the Connection interface **707**, and finally by implementing the Rowset **709** or results mechanism. In each case the Properties, Capabilities and Message files are refined based on the implementations of the Connection and Rowset Interfaces respectively **708, 710**. This process is explained in more detail in the following description of an equivalent example in the preferred embodiment.

As a driver developer, all one needs to do is write the code that is specific to the data source and the functionality that you wish to expose through the DDO API.

In this section, it I shown how to write a driver by example. We will write a DDO driver that provides access to data in flat files. The files are stored in folder. The data in the files is comma-delimited and values are optionally enclosed in quotes. The first line in the file provides the names of the fields. These files are commonly called CSV files and many applications can read and write CSV files.

Any DDO application will be able to use the DDO CSV driver that we will develop together. For the application, the CSV data source would look just like any other data source.

To write a DDO driver,one follows these steps:

- Create the necessary properties, capabilities, and message files.
- Implement the DataSource interface.
- Implement the Connection interface.
- Write code that implements the Rowset interface for data returned by our driver.
- If the data source supports procedure calls, then implement the call results interface.
- If the data source supports transactions, then implement the transaction interface.

In our example, we will show the first four steps. The CSV driver will not implement procedure calling or transactions.

### *Step 1: Create the Properties, Capabilities, and Message Files*

We start by deciding on a package name for our driver implementation. The name will be "demo.csv". This is important because DDO uses the name of your classes to locate its property files. Our DataSource implementation class will be called CSVDataSource.

#### Creating the Properties Files

Based on our package name and the DataSource class name, the properties files will be named as follows:

| File Type | File Name |
|---|---|
| Property values | demo_csv_CSVDataSource_Properties.properties |
| Property descriptions | demo_csv_CSVDataSource_PropertyDescriptions.properties |

DDO allows your driver to inherit properties and property descriptions from DDO. The inheritance is implemented using the class and interface hierarchy. This can save you most of the work. Specifically, you will inherit from the "com_sqribe_access_DataAccess_Properties.properties" and "com_sqribe_access_DataAccess_PropertyDescriptions.properties" files. These files define the logon property and its user and password properties. These files must reside in a folder called "properties" on the Java CLASSPATH.

We create an empty file for the property values because our driver has no special properties.

For the property description, we need to modify the logon property. The default logon property had two properties associated with it, "user" and "password". Ours will have none because it will access CSV files on a local drive. Here is the property descriptions file ("demo_csv_CSVDataSource_PropertyDescriptions.properties"):

```
# override the default logon property
logon.Name=Logon
logon.Description=Logon properties.
logon.Indices=
logon.Required=false
```

By setting "logon.Indices" to none we eliminate the logon attributes.

**Creating the Capabilities Files**

Based on our package name and the DataSource class name, the capabilities files will be named as follows:

| File Type | File Name |
|---|---|
| Capability values | demo_csv_CSVDataSource_Capabilities.properties |
| Capability descriptions | demo_csv_CSVDataSource_CapabilityDescriptions.properties |

Our driver will inherit the contents of the corresponding files in DDO, "com_sqribe_access_DataAccess_Capabilities.properties" and "com_sqribe_access_DataAccess_CapabilityDescriptions.properties" files. These files define common capabilities such as interfaces, command, call, selector, and transaction.

Since we inherit these capability descriptions, our capability descriptions file will therefore be empty. We also inherit default values for these capabilities. For now, this capabilities file will remain empty as well.

**Creating the Messages File**

Your driver may want to throw DataAccessExceptions with specific error messages, and it may want to have other localizable text. These can be leveraged the facilities that are provided with DDO? By doing so you can have DDO lookup the message for you. You will be able to reuse messages that are stored in the DDO message files. Moreover, you will be able to use the interactive tools that come with the DDO SDK to manage your message file.

The DDO message facility uses the class hierarchy to locate messages. It works by looking for a message file for the class that calls it. If the message file is not found, or if the specific message is not found in that file, then it goes on to look in the message files that correspond to super-classes of the specified class. It also looks at interfaces that the class implements and follows that hierarchy as well.

One can have all the messages for all the classes in your driver in one file by having all the classes implement the same interface. Moreover, that interface can extend the DDO Access and Util interfaces. By doing so, if your message is not found in your file, then DDO will go on to look for the message in its own files. In the section "The CSV Interface" below the creation of this common interface will be shown.

The message file will be named "demo_csv_CSV.properties" and will be located in a "msgs" folder on the Java CLASSPATH. The message file will contain entries for messages that are specific to the CSV driver. Consider this example of the file:

```
# The number of column headings doesn't match the number of data columns
# 0 class name
# 1 method name
# 2 heading count
# 3 data count
HeadingCountMismatch.text={0}.{1}(): The number of heading columns, \
{2}, does not match the number of data columns, {3}.

# End of file encountered while looking for end quote
# 0 class name
# 1 method name
# 2 "file" name
# 3 line where quoted string started
MissingQuote.text={0}.{1}(): End of file encountered while \
searching for ending quote; file= {2}, line={3}.

# Unknown column name
# 0 class name
# 1 method name
# 2 column name
InvalidColumnName.text={0}.{1}(): Unknown column name, \
{2}, in selector specification.
```

The file has the same format as property files. The lines that begin with "#" are comments. The entries have the form "<message id>.text". The Message ID is a unique key used in the code to identify a message. The Message ID is not displayed and does not need to be localized. The message text may contain special markers for variables. These markers use are denoted by "{0}", "{1}", etc. By convention, every

message will have at least the first two markers. These will hold the value of the class name and method name where the message is generated. If a long message is broken over multiple lines, you must place a backslash at the place where the line breaks.

The messages in the file will be used in the implementation code that we will develop in this chapter.

**The CSV Interface**

DD uses the class hierarchy to locate messages and properties. It works by walking up the class hierarchy, looking at the message or properties file that corresponds to each class and continuing up if the file or the entry is not found. It also walks up the interface hierarchy for the interfaces that your class implements.

The first Java file in our driver will be the CSV interface. This interface is empty, it is simply used for marking the classes that implement it so that messages and properties can be found. The CSV interface implements the DDO Access and Util interfaces. By doing so it directs the message and property facilities to look at the DDO files if a message or property is not find in your driver's files.

```
package demo.csv;

import com.sqribe.access.*;
import com.sqribe.comutil.*;
/**
 * CSV is a holder interface, allowing properties, messages, etc,
 * common to the csvacc package to be specified once, e.g.,
 * demo_csv_CSV.properties.
 * @see com.sqribe.comutil.PropertySheet
 * @see com.sqribe.comutil.PropertyDescription
 * @see com.sqribe.comutil.Msg
 */
public interface CSV extends Util, Access {
}
```

***Step 2: Implement the DataSource Interface***

In the section "Managing Data Sources" we saw the attributes that define a data source: name, description, class, required Java and native libraries, and the connection string. For CSV, the class attribute will be "demo.csv.CSVDataSource". This is the name of the DataSource implementation we are about to write. The lib and load attributes will be empty as we don't need to load any Java or native libraries for our CSV driver. Lastly, we will define the connection string attribute to point to a folder on the disk where the CSV

files are found. Consider an example for a registry entry for a data source "DataFiles" that will use our CSV driver:

```
DataFiles.desc=Current and historical sales data
DataFiles.class=demo.csv.CSVDataSource
DataFiles.lib=
DataFiles.load=
DataFiles.conn=c:\\data
```

One is now ready to start writing the implementation code.

```
package demo.csv;

import com.sqribe.access.*;
import com.sqribe.comutil.*;

public class CSVDataSource extends DataSourceAdapter implements CSV {
   static final String classname = CSVDataSource.class.getName();

   public CSVDataSource(String pName, String pDesc, String pConn) {
      super(pName, pDesc, pConn, classname);
   }

   public Connection open() throws DataAccessException {
      String folder = getConnectString();
      AccessIO io = AccessIO.createAccessIO("File", folder);
      return new CSVConnection(io, getPropertySheet());
   }
}
```

Our CSVDataSource class implements the DataSource interface by extending the DataSourceAdapter class. The DataSourceAdapter class is an abstract class that provides implementation for most of the DataSource interface. One only needs to implement the constructor and the "open()" method. The DataSourceAdapter provides adequate implementation for all the other methods of the DataSource interface.

Our class defines a static data member called "classname" This is used throughout all the classes in the driver. It is used by the message facility to display the class name that generated the message.

The constructor does nothing special but simply hands off its arguments to the constructor of its super class.

The "open()" method creates a new connection to our data source. It retrieves the folder name (e.g. "c:\data") from the registry (from the DataSourceManager) by calling the "getConnectionString()" method. This method is part of the DataSource interface and is already implemented for us in DataSourceAdapter.

To access files on the disk we use the AccessIO class. This class is part of DDO. It provides IO access to the file system as well as other storage systems. In version 1 of DDO only file IO is supported, but you can imaging how the same mechanism can be extended to allow access to remote files over HTTP or other transport mechanisms.

To open a connection we instantiate an AccessIO object using the "File" protocol and our folder name. The "AccessIO.createAccessIO()" method will create a file AccessIO object and validate that the folder is accessible. We then pass this object on the constructor to our connection object. The next task is to write the CSVConnection class.

### *Step 3: Implement the Connection Interface*

The connection interface is the main interface in DDO. One will need to implement several methods. Start with a few methods. Add implementation for more methods as you continue.

```
package demo.csv;

import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.*;

public class CSVConnection extends ConnectionAdapter implements CSV {
   private static final String classname = CSVConnection.class.getName();
   private AccessIO dir; // folder with CSV files
   private static final String fileTerm = "Table";

   public CSVConnection(AccessIO pDir,
                        PropertySheet pSheet) throws DataAccessException {
      super(pSheet);
      dir = pDir;
      if (dir.isLeaf())
         DataAccessException.rethrow("NotDirectory",
            new Object[] { classname, "CSVConnection", pDir.getName() });
   }
   private void createSchemas() throws DataAccessException {
      Schemas schemas = new Schemas();
      Vector names = dir.listDir();
      for (int idx=0; idx < names.size(); idx++) {
         String name = (String)names.elementAt(idx);
         if (dir.isLeaf(name))
            schemas.add(new SchemaObject(null, name, "", fileTerm, this));
      }
      setSchemas(schemas);
   }
```

```
    public Schemas getSchemas() throws DataAccessException {
        createSchemas(); // refresh the list of objects
        return getSchemasRoot();
    }
    public void close() {
        dir.close();
    }
}
```

The CSVConnection class implements the Connection interface by extending ConnectionAdapter. This abstract class provides default implementation for many of the methods of Connection. It also provides useful helpful functions that aid in the implementation of the CSVConnection class.

CSVConnection has a private data member, "dir", that holds a reference to the AccessIO object representing the folder on the disk that holds the CSV files.

The CSVConnection constructor calls the constructor of ConnectionAdapter. That constructor takes the DataSource property sheet and copies its entries into the connection property sheet. It then validates the folder name where the CSV file exists. This is done by checking that it is not a regular file, i.e., it must be a folder.

Here we see an example of throwing a DataAccessException using its static "rethrow()" method. This method is used to re-throw exceptions in the driver (such as database, IO, and network exceptions) and turn them into a DataAccessException. "rethrow()" is also used to throw a new exception as demonstrated here. The "rethrow()" method takes a message ID as its first argument. The "NotDirectory" message is already defined in DDO. One can use it here without having to define it in your message file.

The next argument to the "rethrow()" method of DataAccessException is an array of objects (typically String objects) that become part of the message. In this example, we include the name of the CSVConnection class and the "CSVConnection" constructor as well as the name of the folder.

The "createSchemas()" method generates the top-level (root) metadata (Schemas). In our driver this is the list of the CSV files in the folder. Note that we don't assume that the CSV file have a ".csv" file-name extension. Instead we assume that all the files in the folder are CSV files. The code for "createSchemas()" is repeated below.

```
private void createSchemas() throws DataAccessException {
   Schemas schemas = new Schemas();
   Vector names = dir.listDir();
   for (int idx=0; idx < names.size(); idx++) {
      String name = (String)names.elementAt(idx);
      if (dir.isLeaf(name))
         schemas.add(new SchemaObject(null, name, "", fileTerm, this));
   }
   setSchemas(schemas);
}
```

Allocate an empty Schemas object. Then obtain the list of CSV files in the folder using the "listDir()" method of AccessIO. For each file name in the list we check that it is a file (and not a folder), and then create a SchemaObject instance for it and add it to the Schemas object. When done, call "setSchemas(schemas)". The "setSchemas()" method is a method of the ConnectionAdapter class. It registers the top-level Schemas object. This is the object that is returned to the application when it calls "getSchemas()" on the Connection.

Override "getSchemas()" in our code to generate the root Schemas. The way it is implemented, each time the application calls "getSchemas()" list the directory again. The effect is that the list of objects in our data source gets refreshed. If a CSV file was added to the folder, a call to "getSchemas()" will add it to the metadata.

The "close()" method is called by the application to close the connection. The ConnectionAdataper implementation of "close()" does nothing. We override it here to close the AccessIO object. In the case of a folder, "close()" actually does nothing because there is no file to close. Nevertheless, we chose to demonstrate this cleanup as a good practice for "close()".

**Providing Column Metadata**

Information about columns, their name and type, is derived from the CSV file itself. For that purpose, we write a class called CSVFile. This class encapsulates the implementation of reading a CSV file. It deals with reading lines, dealing with delimiters, obtaining column names from the first line, sampling the data to guess the type of the column by looking at the next 5 lines, and finally reading the data. The source for CSVFile is listed later in this chapter. For now we must focus on the DDO part of things which is creating the SchemaObjectColumns and implementing the "getSchemaObjectColumns()" method of Connection.

Before we continue, let's review the methods of the CSVFile class that we will be using:

| Method | Description |
|---|---|
| Constructor | Creates a new CSVFile object for this AccessIO. |
| getLineTokens | Peels off the delimiters and returns the items on a line. |
| getSampleData | Returns tokenized data for the first few lines in the file. |

To keep things simple, our driver only supports three data types: date, number, and text. Now let's add implementation for "getSchemaObjectColumns()" to our CSVConnection class:

```
        public SchemaObjectColumns getSchemaObjectColumns(
                SchemaObject pSchema) throws DataAccessException {
❶       CSVFile csvfile =
              new CSVFile(dir.createAccessIO(pSchema.getName()));
❷       Vector headingsLine = csvfile.getLineTokens(); // Get the headings
❸       Vector dataLine[] = csvfile.getSampleData(); // Sample data
        csvfile.close();
        int headingsCount = headingsLine.size();
        int dataCount = dataLine[0].size();
        if (headingsCount != dataCount) {  // Must be the same number
           DataAccessException.rethrow("HeadingCountMismatch",
              new Object[] { classname, "createMetaData",
              new Integer(headingsCount), new Integer(dataCount) } );
        }
❹       SchemaObjectColumns columnMetaData = new SchemaObjectColumns();
        for (int idx = 0; idx < headingsCount; idx++) {
           String name = (String) headingsLine.elementAt(idx);
           int prev = 0, curr = 0;  // type of previous and current value
           int size = 0;            // size of this coulum
           int scale = 0, prec = 0; // precision and scale (numeric)
           for (int i = 0; i < dataLine.length; i++) {
              if (dataLine[i] != null) {
❺                String sample = (String) dataLine[i].elementAt(idx);
                 Field field = getField(sample);
                 curr = field.getType();
                 if (prev != 0 && prev != curr)
                     curr = Field.Text;
                 prev = curr;
                 size = Math.max(size, sample.length());
                 if (curr == Field.Decimal) {
                    scale = Math.max(scale,
                          ((DecimalField)field).decimalValue().scale());
                    prec = size;
                 }
              }
           }
```

```
❻       SchemaObjectColumn soc = new SchemaObjectColumn(
                    pSchema, name, curr, curr,
                    getDBTypeName(curr),
                    size, prec, scale, "");
        columnMetaData.add(soc);
    }
    return columnMetaData;
}
```

The method signature for "getSchemaObjectColumns()" is defined in the Connection interface and in the base implementation class ConnectionAdapter. We override the method to provide appropriate implementation for CSV files.

The purpose of the "getSchemaObjectColumns()" method is to describe the columns of a data object—a CSV file. This includes the number of columns, their name, type, and size. We start in step ❶ by constructing an AccessIO object for the object represented by the "pSchema" argument. The AccessIO object will allow us to read the file. We then construct a CSVFile object for this file. The CSVFile object will allows us to parse the CSV file. The implementation source-code for the CSVFile class is provided in the section "The CSVFile Class" later in this chapter.

Now that we have a CSVFile object for our CSV file, we read the heading line (the first line in the CSV file contains the column headings). See step ❷. These headings are the column names. In step ❸ we read "sample" data—the first 5 data lines in the file. We use this data to guess the data type of each column. CSV files really do not have type information. By looking at the first few values of each column we can see of they are all valid dates or numbers. Otherwise we take the column as text.

We validate that the number of headings matches the number of values in the data lines (we compare the number of headings to the number of items on the first line of data). If they don't match, we throw an exception. The Message ID used here "HeadingCountMismatch" refers to an entry in the message file "demo_csv_CSV.properties".

In step 4 we construct a new SchemaObjectColumns object. This object will hold the collection of SchemaObjectColumn objects that describe the columns of our CSV file.

Next, we process columns one by one. We go over the sample data for each column to determine the type of the column. This is done using the "getField()" method (see below). The "getField()" method returns a typed field (DateField, DecimalField, or

TextField). We get the type of the field using the "getType()" method of the Field class. We compare that type against the type of previous columns. If there's a conflict, then we resolve this column to be a text column. However, if all the values in the column are valid date values, we resolve this column to be a date column. We apply the same rule for columns with all values being valid numeric values.

We use the sample data to determine the size of the column as well as precision and scale for decimal columns.

Once we have the column name, type, and size, we can construct the SchemaObjectColumn for it. This is done is step ❻. We set the parent attribute of the column to point to the data object (the csv file SchemaObject), next we pass the name and the type. Our driver uses the DDO field type as the database type as well. The "getDBTypeName()" method (see below) provides a descriptive name for the type. The SchemaObjectColumn is added to the SchemaObjectColumns variable.

Next, let's look at the "getField()" and "getDBTypeName()" methods.

```
private Field getField(String sample) {
   Field field;
   field = new DateField(sample);  // see if valid date value
   if (field.isNull()) {
      field = new DecimalField(sample); // see if valid number
      if (field.isNull())
         field = new TextField(sample); // default is text
   }
   return field;
}
private String getDBTypeName(int pType) {
   String typeName;
   switch(pType) {
      case Field.Number:
         typeName = "NUMERIC";
         break;
      case Field.Date:
         typeName = "DATE";
         break;
      default:
         typeName = "VARCHAR";
         break;
   }
   return typeName;
}
```

The "getField()" method returns a typed field for the given value. It starts by attempting to construct a DateField from the given value. If the value is not a valid date this would fail and the value of the DateField will be null. In such case we proceed to

try a DecimalField. If that fails, we take the value as a TextField. The function returns the typed field object.

The "getDBTypeName()" method returns a name for the type of the field. DDO does not define the DBTypeName. This is at the discretion of the data source. We return "NUMERIC" for a decimal number, "DATE" for a Date field, and "VARCHAR" for anything else.

**Implementing getData**

To complete the implementation of the Connection interface in our CSVConnection class, we need to implement the "getData()" methods. Let's start by looking at the simpler "getData()" that takes a SchemaObject argument. In the following section we will discuss the implementation of "getData()" with a Selector.

Rowset is an interface for processing a result set. The "getData()" method returns a Rowset interface. The driver must implement this interface. In our example, the CSVRowset implements the Rowset interface for processing a CSV file. Since most of the logic is in the CSVRowset, our "getData()" implementation is easy.

```
public Rowset getData(SchemaObject schema) throws DataAccessException {
   CSVFile csvfile = new CSVFile(dir.createAccessIO(schema.getName()));
   return new CSVRowset(csvfile, getSchemaObjectColumns(schema));
}
```

We construct a CSVFile object that will be used for reading the data from the file. The CSVRowset class will use the CSVFile object to read the data. The CSVRowset constructor will takes two arguments. The CSVFile object and the column metadata (a SchemaObjectColumns) for this object.

In the section, "Step 4: Implementing Rowset", we will see how the CSVRowset implements the Rowset interface for processing results of "getData()".

**Implementing getData with Selector**

A Selector provides a more flexible means of retrieving data from an object. In the current version of DDO, the selector allows the application to specify the desired columns and their order. In future versions of DDO, the selector may be used to specify filtering, sorting, and join criteria as well.

DDO provides a class RowsetFilter that applies a selector to a Rowset to yield a new Rowset. The new Rowset will have the fields specified in the Selector and in the order that is specified in the Selector.

The ConnectionAdapter class provides a default implementation for "getData(Selector)". The implementation uses the RowsetFilter to apply the selector to the Rowset that the simpler "getData()" returns. The code in ConnectionAdapter looks like this:

```
public Rowset getData(Selector selector) throws DataAccessException {
   return new RowsetFilter(getData(selector.getObject()), selector);
}
```

For our CSV driver, this implementation is satisfactory. Therefore, we do not need to provide an implementation for "getData(Selector)", the base implementation will do.

Before we proceed to "Step 4: Implementing Rowset", let's examine the code for the CSVFile class. This class encapsulates reading and parsing a CSV file.

**The CSVFile Class**

The CSVFile class was using in the implementation. We give the source here for your reference.

```
package demo.csv;

import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.Vector;
import java.io.IOException;

public class CSVFile implements CSV {
   private static final String classname = CSVFile.class.getName();
   private AccessIO access;
   private String delimiters = "\",";

   public CSVFile(AccessIO pAccess) {
      access = pAccess;
   }
```

```
public Vector getLineTokens() throws DataAccessException {
   String line = readLine();
   Vector tokens = null;
   if (line != null) {
      int start, end;
      tokens = new Vector();
      for (start=0; start < line.length(); start=end+1) {
         end = getDelimiter(line, start);
         if (start != end) {
            String token = line.substring(start, end);
            if (token.length() != 0) tokens.addElement(token);
         }
         if (end < line.length() && line.charAt(end) == '\"') {
            line = line.substring(++end);
            start = 0;
            end = getQuotedString(line, start);
            if (start<=end) {
               tokens.addElement(line.substring(start, end++));
            } else {
               DataAccessException.rethrow("MissingQuote",
                     new Object[] { classname, "getLineTokens",
                                    access.getName(), line } );
            }
         } else if (start==end) {
            tokens.addElement(null);
         }
      }
   }
   return tokens;
}
public Vector[] getSampleData() throws DataAccessException {
   Vector[] dataLine = new Vector[5];
   int idx;
   for (idx=0; idx<dataLine.length; ++idx) {
      dataLine[idx] = getLineTokens();
      if (dataLine[idx]==null) break;
   }
   return dataLine;
}
private int getDelimiter(String pLine, int pIdx) {
   int idx = pIdx;
   for (; idx<pLine.length(); ++idx) {
      char c = pLine.charAt(idx);
      if (delimiters.indexOf(c) != -1) break;
   }
   return idx;
}
```

```
    private int getQuotedString(String pLine, int pIdx) {
       int              idx;
       if ((idx=pLine.indexOf('"', pIdx))==-1) {
          try {
             String line = readLine();
             if (line!=null) {
                idx = pLine.length();
                pLine.concat(line);
                idx = getQuotedString(pLine, idx);
             }
          } catch (DataAccessException e) { }
       }
       return idx;
    }
    private String readLine() throws DataAccessException {
       StringBuffer line = new StringBuffer();
       while(true) {
          int token = read();
          if (token == -1) break;
          if (token=='\r') {
              token = read();
              break;
          }
          line.append((char)token);
       }
       String ret = line.toString();
       if (ret.length()==0) ret = null;
       return ret;
    }
    private int read() throws DataAccessException {
       int token = 0;
       try {
          token = (access.accessReader()).read();
          if (token == -1) access.closeReader();
       } catch (IOException e) {
          DataAccessException.rethrow("IOError",
             new Object[] { classname, "read", e.toString() } );
       }
       return token;
    }
}
```

### *Step 4: Implementing Rowset*

The Rowset interface is a key interface in DDO. The way you implement this interface will have great implications for the performance of your driver. Before we dive into the implementation of our CSVRowset class, let's review some of the theory behind the Rowset interface and its implications on performance.

- Rowset is an interface. You are expected to provide an implementation that optimizes performance for data retrieval from your data source.

- The Rowset interface is designed to support very large result sets. You should not hold the entire result set in memory unless you can be sure that the result set is a single row or contains very few rows. DDO provides an implementation of Rowset, VectorRowset, that uses a Java Vector. You can use it for the case of a single row or very few rows. However, you should not use the VectorRowset class if you are retrieving a result set.
- Your implementation of Rowset can hold off fetching the data until it is actually requested via the "next()" method. Take advantage of this to improve performance by fetching rows just in time. Of course, your application can perform some "fetch ahead" or buffering, but it should not retrieve all the data up front.
- The application using Rowset is not required to fetch all the data. The "close()" method signals that the application will not be fetching any results that are still outstanding. Your implementation should respect the "close()" method. If your data source requires that all the data be process, you can silently skip the data in your driver, rather than force the application to retrieve all the data.
- To minimize object creation, you can—and should—use the same record and the same fields when "next()" is called. Rather than allocate new fields, the driver can respond to "next()" by populating the same fields with new data. If the application wants to hold references to multiple records, it is its responsibility to make copies of rows. For most applications, this is not required, so why do all the extra work?

With this in mind, we are ready to review the CSVRowset implementation. Our implementation will implement the Rowset interface. We will provide methods that implement all the methods of the Rowset interface. We will hold a single row in memory and populate it with new values in the "next()" method.

Here is the code:

```
package demo.csv;

import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.*;

public class CSVRowset implements CSV, Rowset {
   private static final String classname = CSVRowset.class.getName();
   private CSVFile csvfile;
   private SchemaObjectColumns soc;
   private VectorRow currentRow;

   public CSVRowset(CSVFile pCSVFile, SchemaObjectColumns pSoc)
                                        throws DataAccessException {
      csvfile = pCSVFile;
      soc = pSoc;
      currentRow = new VectorRow();
      allocateFields();
      csvfile.getLineTokens();  // Position past the headings
}
   public Row getRow() {
      return currentRow;
   }
   public int getFieldCount() {
      return currentRow.getFieldCount();
   }
   public Field getField(int index) throws DataAccessException {
      return currentRow.getField(index);
   }
   public Field getField(String name) throws DataAccessException {
      return currentRow.getField(name);
   }
   public void close() {
      try {
         csvfile.close();
      } catch (Exception e) { }
   }
```

The constructor takes the two arguments,

| Argument | Description |
|---|---|
| CSVFile pCSVFile | Provides an abstraction of a CSV file. Allows us to parse the file and obtain the data it holds. |
| SchemaObjectColumns pSoc | Column metadata for this CSV file. Provides column names, size, and type. |

The constructor creates a Row to hold one record. We use the VectorRow class in DDO. This class provides a simple implementation of a Row that uses a Java Vector to hold the fields. Your driver may provide its own implementation of the Row interface.

Next we allocate the fields based on the column metadata. This is done in the "allocateFields()" method. We'll see that shortly. We read the first line from the CSV file.

This line has the column headings. We don't need that as we already have this information in the column metadata. We simply skip this first record.

After the constructor you see the "getRow()", "getFieldCount()", and "getField()" methods. The implementation of these methods is quite simple. The "close()" method will close the AccessIO object and close the file. The Rowset interface does not allow for errors during "close()" so we silently ignore any errors during the "close()" method.

Now let's see the "next()" method.

```
public boolean next() throws DataAccessException {
   Vector tokens = csvfile.getLineTokens();
   if (tokens==null) return false;  // end of file reached
   for (int i=0; i < getFieldCount(); i++) {
      String value = (String)tokens.elementAt(i);
      Field f = getField(i);
      if (value.length()==0) f.setNull(true);
      else f.setValue(value);
   }
   return true;
}
```

We start by reading a line from the CSV file. We get the line broken into fields according to the number of fields and their order in the file. We then go through the fields. We set the value of appropriate field to either null (if the file has no value for this field) or to the actual value using the "setValue()" method of Field. Note that Field may by a DecimalField, DateField, or TextField, the appropriate "setValue()" will be called (this is the essence of polymorphism).

"next()" returns true when we process a record and false when no more records are available and the end of the file has been reached.

Now let's see the "allocateFields()" method.

```
private void allocateFields() throws DataAccessException {
   Enumeration enum = soc.elements();
   SchemaObjectColumn col;
   while (enum.hasMoreElements()) {
      col = (SchemaObjectColumn)enum.nextElement();
      currentRow.addField(Field.createField(col.getName(),
                          col.getFieldType(), col.getSize(), true));
   }
}
```

"allocateFields()" goes through the column metadata and allocate a field for each column. The "createField()" method of the Field class creates a typed field based on the

specified field type (second argument). In our case it will return one of DateField, DecimalField, or Text Field. We then add the field to our Row.

As demonstrated in our example, writing a simple driver that provides schemas, objects, column metadata, and supports "getData()" and Selector is not difficult. To support command execution with parameters, and to support procedure calling you will need to write more code. If your data source is Multidimensional, you should also support a MDSelector and provide additional metadata via the MDSchemaObject interface.

BEST MODE

The following programming considerations and descriptions of related tools and common facilities are described in the interests of describing the best mode of practicing the invention known to Applicants at this time.

***Driver Organization Tips***

Use the DDO adapters to provide default behavior. The DataSourceAdapter and ConnectionAdapter, for example, provide default methods for features that your driver does not implement. They also provide an array of helper methods to perform common functions.

For simple drivers, most of the operational methods can be placed in the Connection class. For complex drivers, the Connection class may become too large and complex. In this case, delegating the functions to specialized worker objects will make the driver easier to understand and maintain. One may want to add specialized worker classes for object or procedure processing. Or, one may want to handle the formation of the metadata hierarchy in a specific class.

In many respects, the driver will represent a bi-directional gateway. It will present metadata, results, and execution operations to the application, using the DDO interfaces. These operations will be translated to invocation sequences recognized by the data source.

DDO provides a number of tools to assist in application deployment. One such tool is the RegEditor. The RegEditor is a data-driven application. Driver writers should add connection specification information for their drivers to enable configuration through the RegEditor tool. This is done by:

1. Adding the common name and descriptive name of the driver to the DataSources.drivers property in properties/com_sqribe_access_DataSourceManager_Properties.properties. In this property file snippet, we see six drivers identified. The templates for these drivers exist in the corresponding DataSourceManager message file. Each pair is separated by a semicolon (;). The name and description are separated by white space. If description contains white space, it must be enclosed in quotes (""). The strings to build the property descriptions are held in the message file: com_sqribe_access_DataSourceManager.properties. The message file contains resource strings (like those used to create labels for UI dialogs) and message text. The resource strings beginning with the names listed in this property are used to create property descriptions from the pseudo property descriptions for class, lib, load, etc., provided in the com_sqribe_access_DataSourceManager_PropertyDescriptions.properties file.

```
#
# These are the names and their descriptions we will use to build property
# descriptions and entries for the driver templates. These are used by the
# Registry administration tools to provide driver configuration information.
See the API documentation for com.sqribe.access.Registry for more
information.
#
DataSources.drivers= \
csvacc "CSV driver"; \
essacc "Essbase driver"; \
jdbcacc "JDBC driver"; \
msmdacc "Microsoft ADO MD driver"; \
psacc "PeopleSoft driver"; \
sapr3acc "SAP R/3 driver"
```

1. Adding the template description for the driver in the message file msgs/com_sqribe_access_DataSourceManager.properties. This template is for the CSV driver. The template mirrors the information required in the registry data source entries. The difference is in the connection string. The connection string has bracketed (< >) entries for each substitution value in the connection string. The label in the brackets is used as the parameter label in the registry editor. A driver may have multiple connection string templates. These may correspond to multiple lib entries. In the case of the DDO relational database driver, the corresponding entries represent related JDBC driver and connection strings.

```
#
# CSV driver template
#
csvacc.name.string=CSV DataSource Template
csvacc.class.string=com.sqribe.csvacc.CSVDataSource
csvacc.lib.string=
csvacc.load.string=
csvacc.conn.string="CSV:File:<Fully Qualified Directory Path Name>"
csvacc.desc.string=This data source represents a directory tree rooted in a
file system. \
Files in the tree having the file extension, ".csv", are interpreted as
delimiter \
separated files. These files represent objects to this driver.
```

```
#
# JDBC datasource driver template
#
jdbcacc.name.string=JDBC DataSource Template
jdbcacc.class.string=com.sqribe.jdbcacc.JDBCDataSource
jdbcacc.lib.string=sun.jdbc.odbc.JdbcOdbcDriver \
oracle.jdbc.driver.OracleDriver \
com.sqribe.License.SQRIBE970801Ora \
com.sybase.jdbc.SybDriver \
com.sqribe.License.SQRIBE970801MSsqlSybase
jdbcacc.load.string=
jdbcacc.conn.string="JDBC:ODBC:<ODBC DSN>" \
"jdbc:oracle:oci7:@<Oracle TNS Name>" \
"jdbc:Weblogic:@<Oracle TNS Name>" \
"jdbc:sybase:Tds:<Host Name>:<Port Address>/<Database Name>" \
"jdbc:Weblogic:Tds:<Host Name>:<Port Address>/<Database Name>"
jdbcacc.desc.string=This driver provides data source access through JDBC. \
The JDBC driver name, e.g., com.sybase.jdbc.SybDriver, is given in the "lib"
\
resource. Only a single name should be specified. This JDBC driver name must
\
have a corresponding connection specification, given in the "conn" resource.
\
The relative entries of the "lib" and "conn" resource lists are correlated,
e.g., \
the "oracle.jdbc.driver.OracleDriver" corresponds to the \
"jdbc:oracle:oci7:@<Oracle TNS Name>" connection template. \
This DDOdriver provides support to RDBMS tables and stored procedures.
```

### *Messages and Exceptions*

Driver writers are encouraged to place their messages in a single file for the driver package. For example, if your driver was in the package, demo.csv, one could create an empty interface called CSV. Each class in the driver package would implement the CSV interface. As a result, each class would have access to the messages for CSV. Further, the

CSV interface would extend the com.sqribe.access.Access empty interface. Doing this gives the driver access to the Access messages. Since Access extends the com.sqribe.comutil.Util interface, the driver also has access to the common utility messages. The CSV empty interface would look like:

```
package demo.csv;

import com.sqribe.access.*;

/**
 * CSV is a holder interface, allowing properties, messages, etc, common
 * to the csv package to be specified once
 */
public interface CSV extends Access {
}
```

The message facility supports user and log messages. The log messages provide more information, than the user messages. While there is no requirement to have both, and in some situations it does not make sense to have both, it is a good idea to provide both message forms. The message facility provides automatic logging of messages. Hence, when a message is written, the facility looks for a log message pattern and, if so, generates a log entry. User and log messages follow specific conventions. While there is no requirement to follow these conventions, the DDO exceptions, e.g., DataAccessException, provide convenience methods, e.g., rethrow, that anticipate the use of these conventions.

- The first substitution argument is the name of the class requesting the message.
- The second substitution argument is the name of the method requesting the message.
- The remaining substitution arguments, if any, provide specific details for the message.

```
# There are no object names in the from clause of the select statement
# 0 The class reporting the error
# 1 The method reporting the error
# 2 The from clause
ObjectNameMissing.text=There are no object names in the FROM clause: "{2}".
ObjectNameMissing.logtext={0}.{1}(): There are no object names in the \
FROM clause: "{2}".
```

As can be seen, the difference between the user and log message pattern is the prepending of "{0}.{1}(): " for the class and method name. It is not necessary to add message substitutions for a stack trace or exception string. These are available through the rethrow convenience methods.

```
    /**
     * Throw an exception after logging entry
     * @exception UtilException
     *              A general exception for data source
     *                          operations such as
     *              making a connection or performing a command.
     * @see ApplicationLog#logMsg
     */
    public static void rethrow(String pMsgId, Object[] pArgs) throws
            UtilException {
       throw new UtilException((String)pArgs[0], pMsgId, pArgs);
    }

    /**
     * Rethrow exception after logging entry. A stack trace is logged
      * with the message.
     * @param pMsgId       The message identifier
     * @param pArgs     The substitution arguments for the message,
     *                  pArgs[0] is the classname
     * @param pExc      The exception to be recorded
     * @exception UtilException
     *            A general exception for data source
     *              operations such as
     *            making a connection or performing a command.
     * @see ApplicationLog#logException
     */
    public static void rethrow(String pMsgId, Object[] pArgs,
              Throwable pExc) throws UtilException {
       if (pExc instanceof UtilException)
                 throw new UtilException((String)pArgs[0], pMsgId, pArgs);
       else throw new UtilException((String)pArgs[0], pMsgId, pArgs,
                 pExc);
    }
```

Error notification is through exceptions. This allows application developers to organize error handling on the edges rather than in the main code path. Exceptions are often generated using the rethrow methods. Use of the getMsg method in the Message Facility, for example, is when all of the messages pertaining to an operation need to be gathered before raising an exception.

```
Object[] objs = new Object[]
    { classname, "bindParameters", new Integer(0), new
      Integer(parmcount) };
addErrorMsg(ApplicationLog.getMsg().getMsg(classname, "TooManyParameters",
    objs ));
```

### *Properties and Capabilities*

Capabilities are read-only object properties from an application perspective. From a driver perspective, they are object properties. Though they are processed in much the same

way, properties and capabilities are in separate name spaces. This means that a property and capability may have the same name but be treated as distinct entities. The rationale for a separate name space and form is purely pragmatic and is rooted in the perceived usage patterns.

From a driver developer's perspective, an attribute should be stipulated as a capability, if and only if, the application should not change the value.

Descriptions are required for all properties and capabilities. If a capability is a string, is not computed at runtime, and it is felt that a description is not warranted, then one should make it a message string type.

Properties, capabilities, their values, and their descriptions may be localized. This may be accomplished at the granularity of an attribute, i.e., a single property attribute within a property resource bundle may be translated, placing it in a localized property resource bundle. One could translate the property description names, and nothing else. Or, one may choose to translate the name and descriptive name.

Property and capability descriptions may be hierarchical. The hierarchical structure can be used by the application to create property pages. The logon property is a hierarchical structure. The RegEditor tool uses this structure to create a logon panel.

### *Debugging and Testing Your Driver*

Three command line sample programs as well as a graphical test tool are available for driver development. The command line sample programs are located in the sample directory. They are:

**AccessMeta**
This program may be used to test the metadata hierarchy driver functionality. It is a simple program that takes a schema name as an argument and produces an HTML report displaying the metadata for the subtree whose root is the named schema.

**AccessObjects**
This program may be used for drivers supporting the DDO object retrieval interface. Given a qualified object name, this program produces an HTML report displaying the result set for the object.

**AccessProcs**
This program may be used for drivers supporting the DDO call retrieval interface. Given a qualified procedure name and a parameter list, this program produces an HTML report displaying the in/out and output parameters, the return value, and the result sets.

TOOLS AND COMMON FACILITIES

### *Registry Editor*

You use the DDO Registry Editor to manage data sources. Data source management through property file manipulation was described earlier. The DDO Registry Editor provides a graphical interface for data source management.

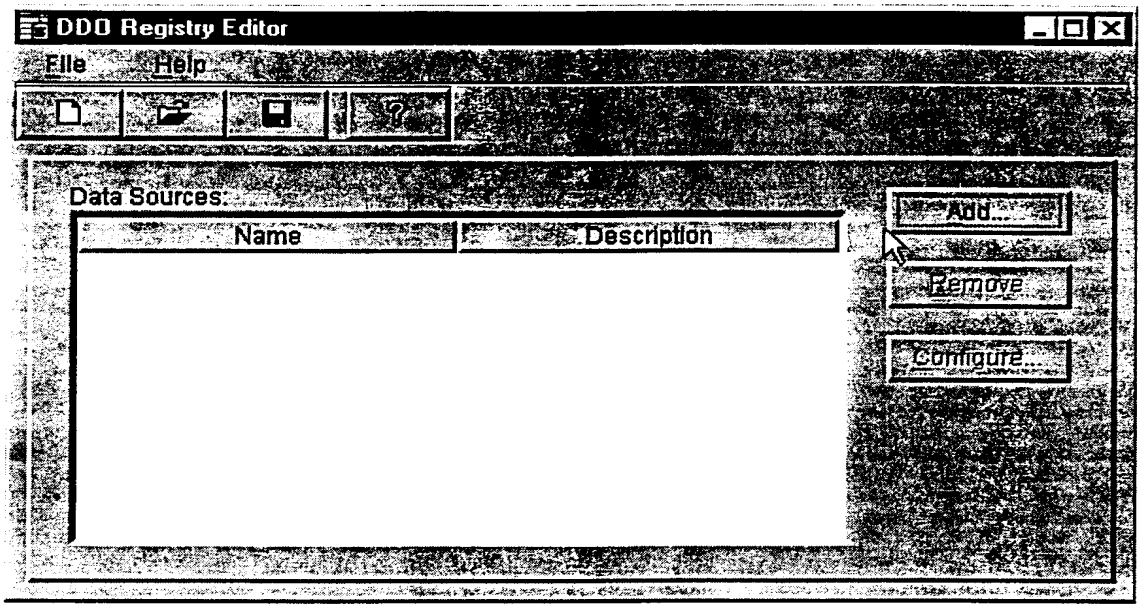
DDO allows multiple registries. Each registry contains a number of data sources. Each data source defines the objects and connection specifications for the source. A data source specification consists of:

- The data source name (required). This is the logical name for this data source. DDO applications will associate this name with the connection.
- The data source descriptive name (optional). The descriptive name is used to assist the user in selecting a data source. DDO applications may display the descriptive name along with the data source name. Alternatively, they may display the descriptive name as a *tooltip*.
- The class name of the DDO data source driver (required). This is the class is loaded by the DDO data source manager when the data source is selected.
- Additional Java classes to be loaded with the data source driver (may be required by the driver). The driver may require additional classes to be loaded, prior to the instantiation of the data source driver class. These are provided by the driver and will appear when the driver is selected in the registry editor. One should not have to provide additional information here.
- Additional JNI packages to be loaded with the data source driver (may be required by the driver). The driver may require platform specific libraries to be loaded, prior to the instantiation of the data source driver class. These are provided by the driver and will appear when the driver is selected in the registry editor.
- The base connection information to be supplied to the driver (optional). While all drivers require connection information, some drivers allow this information to be supplied at the time the connection is requested. Others require some information in

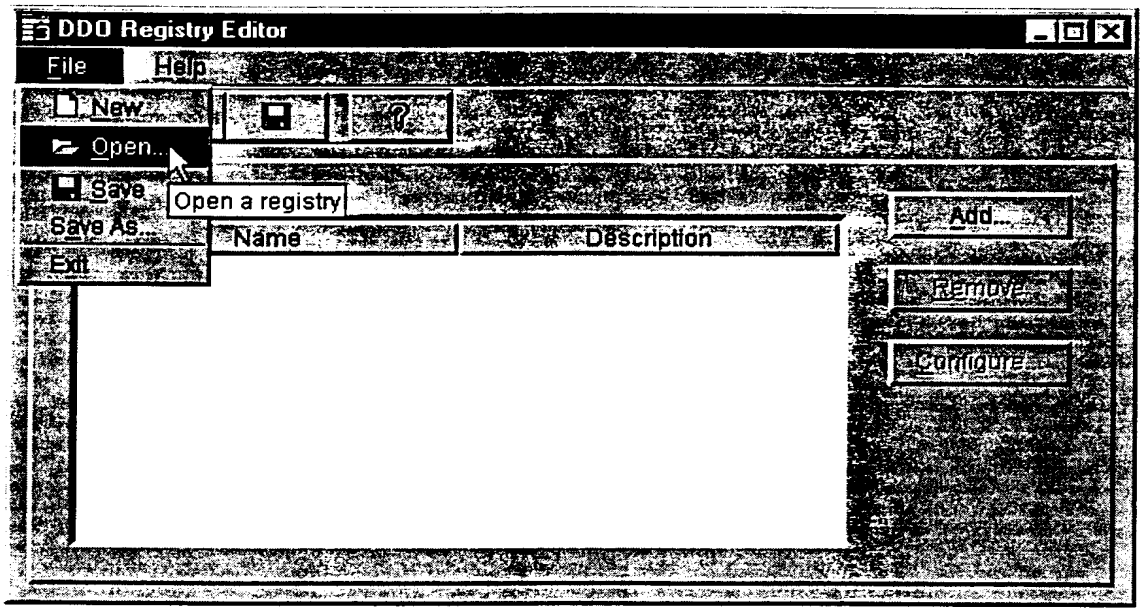
advance. The drivers supply templates for connection information. The Registry Editor uses the templates to provide a dialog for the connection string.

One can quickly create and update registries with the DDO Registry Editor. In addition, the editor provides a test connection function so that you can be sure that the data source specification is correct before trying it in your application.

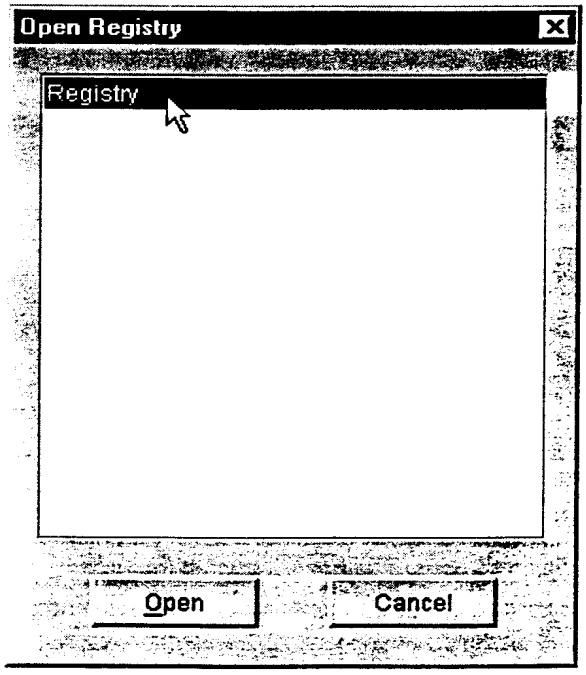
When you start the DDO Registry Editor, you will see the initial dialog window.



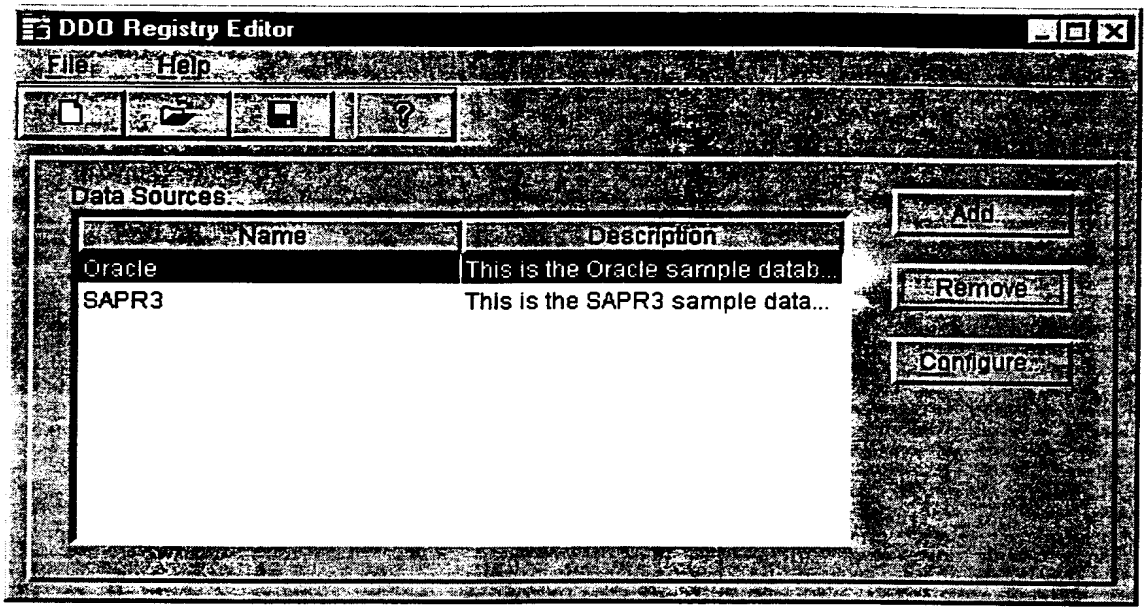

You need to select the **File** menu button, the toolbar new button, or the toolbar open button to update or create a registry.



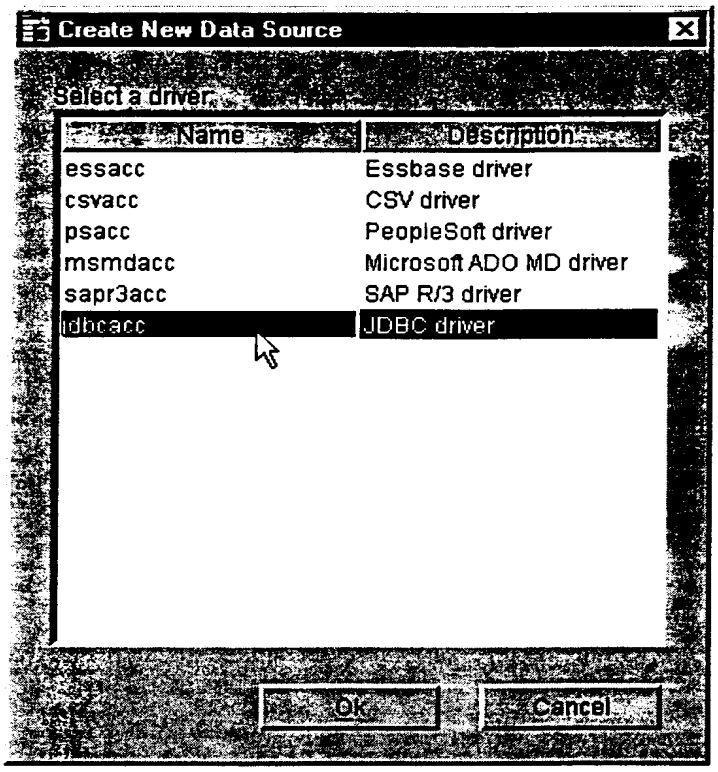

We selected the open option on the **File** menu button. A dialog is presented, containing a list of registries. We have selected the "Registry" registry by clicking on the line item and then clicking on the Open button. This is the default registry and should always appear.



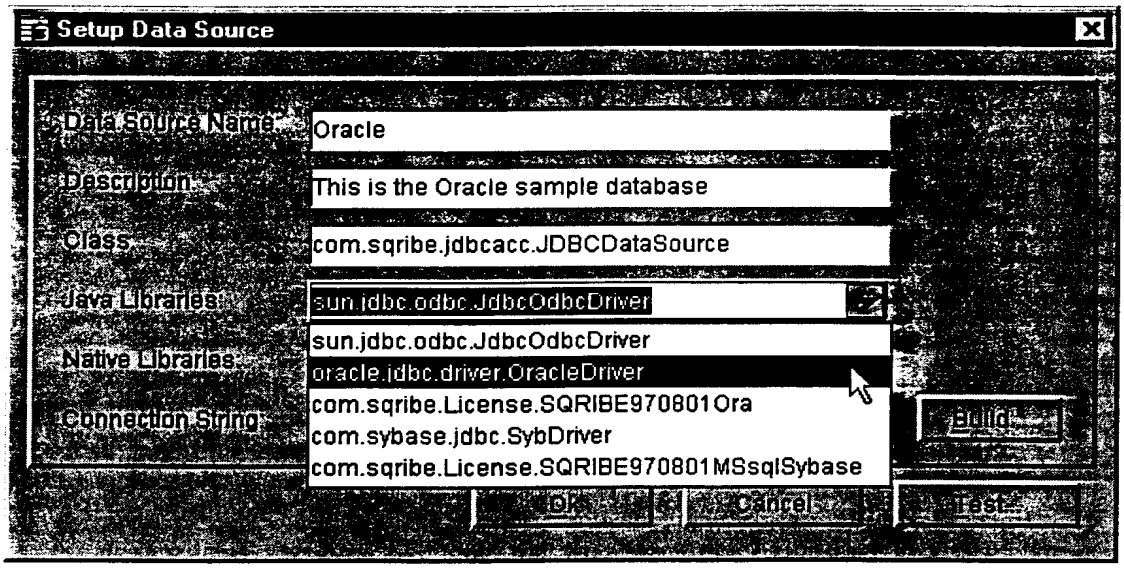
The Open Registry dialog disappears and the data sources defined in the registry appear on the main dialog window. The registry contains a single data source, "SAPR3". We want to add a specification for an Oracle data source.



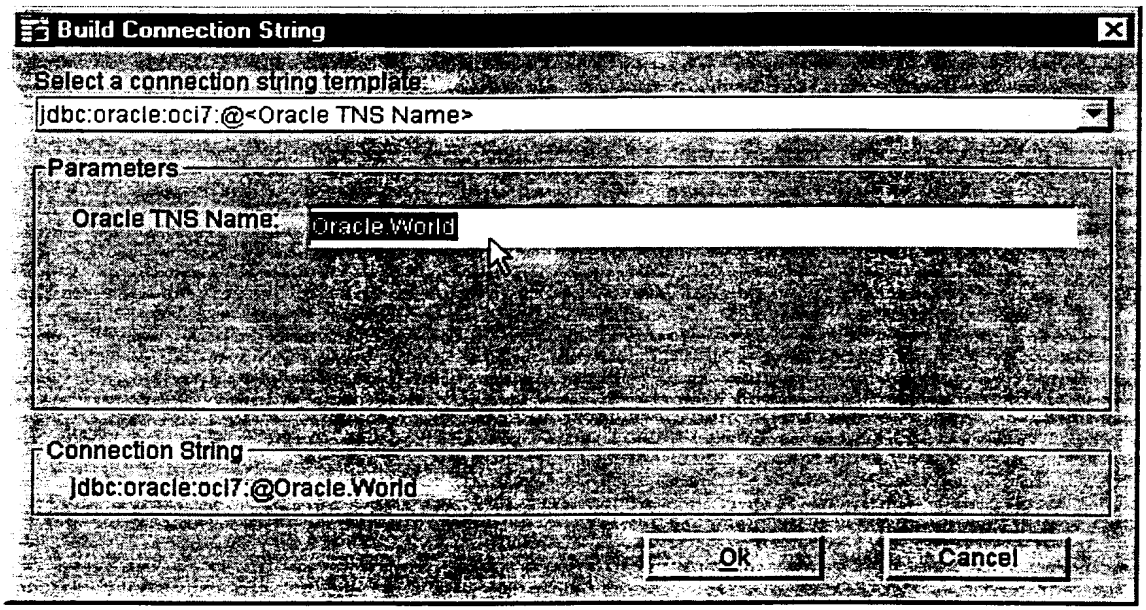

We click the **Add** button and are presented with the Create New Data Source driver selection dialog. This is where you select the kind of DDO driver needed to connect to the data source. Since we want to create an Oracle data source, we click on the "jdbcacc" driver name and click the **ok** button.



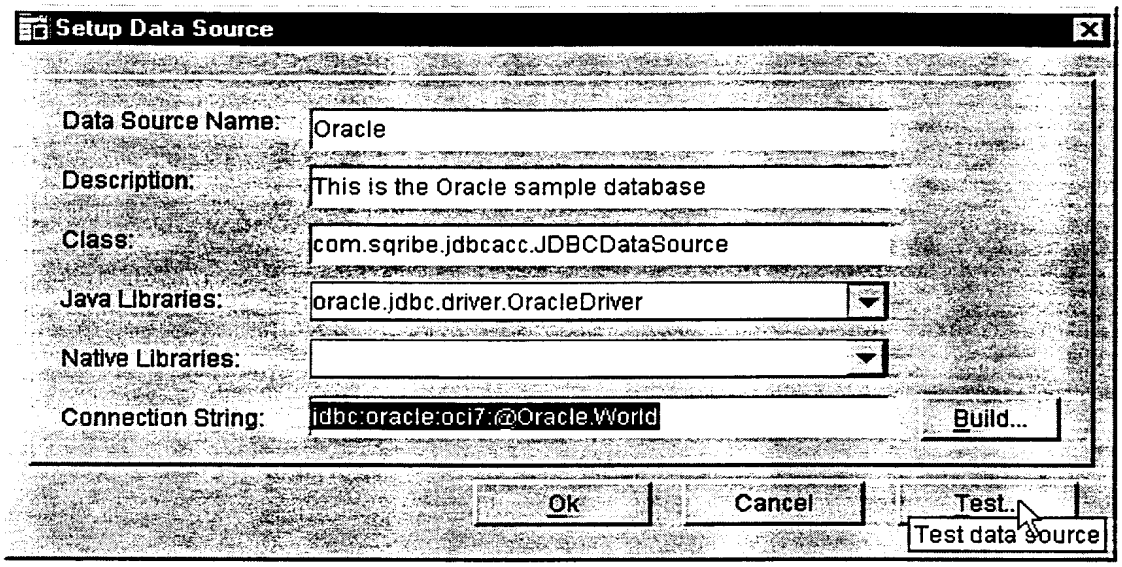

The driver selection dialog disappears and the data source configuration dialog appears. We have named the data source "Oracle" and provided a description. This driver requires Java classes to be loaded. These classes represent different ways to connect to the Oracle data source. The list was presented when we clicked the list box down arrow button on the "Java Libraries" line. We have selected the Oracle JDBC driver. Oracle Corporation provides this driver. The driver may have been installed with Oracle. The current driver may be obtained from the Oracle WEB site. At this writing, the Oracle driver is contained in <Oracle directory>/jdbc/lib/classes111.zip.



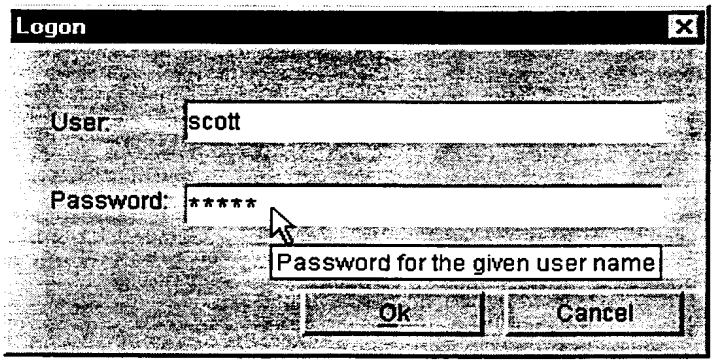

As a result of selecting the Oracle JDBC driver, there were no Native Libraries. Also, choosing the Oracle JDBC driver affects the connection string template. We clicked the **Build** button and were presented with a Build Connection String dialog. We entered "Oracle.World" for the TNS name, which we got from our Oracle TNS file. This template uses the Oracle OCI 7 protocol layering. An alternative would have been the "thin" driver protocol layering.



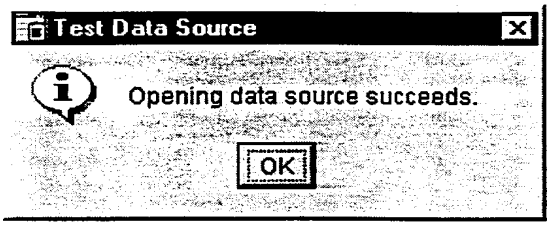
After clicking **Ok**, the Build Connection String dialog disappears and we have returned to the Setup Data Source dialog. The connection string is displayed. We click the **Test** button to insure that the data source specification is correct.



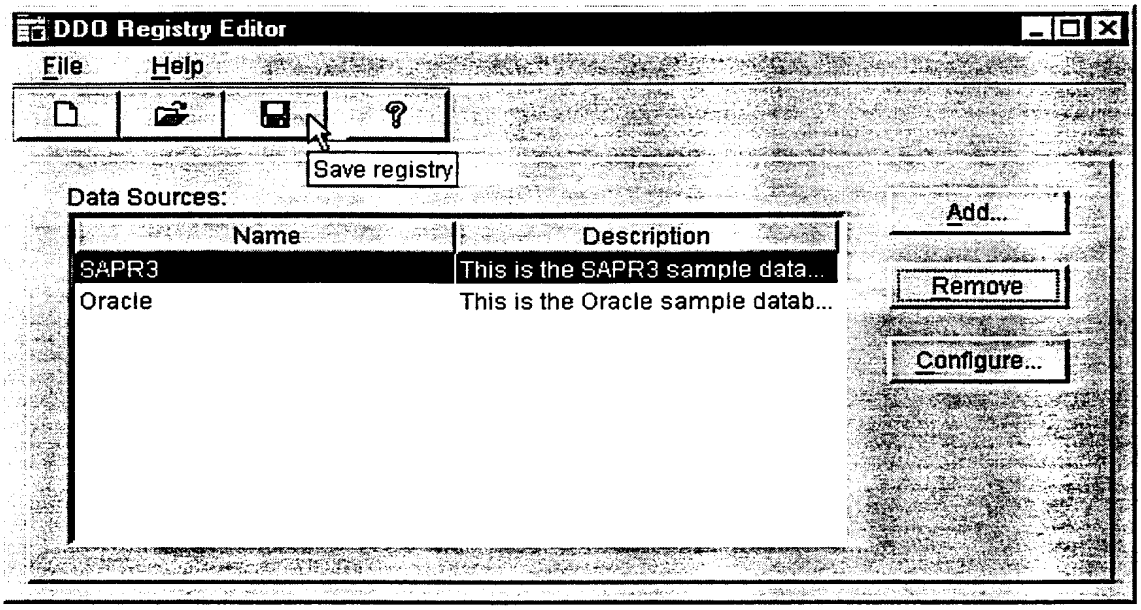

We are prompted with the connection logon dialog. Oracle requires a user name and password for authentication. We have provided this information and clicked okay.



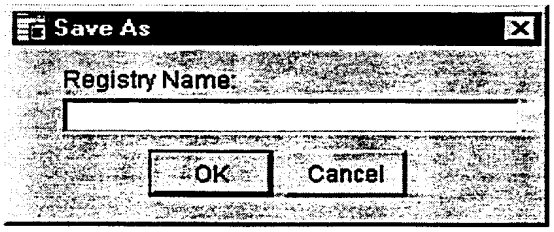

The Logon dialog disappears and a message dialog appears, indicating the connection test was successful. If the data source specification were incorrect, then an error dialog would have appeared instead of the Logon dialog. If the user or password were incorrect, then an error dialog would have appeared after the Logon dialog.




In this case everything was fine. So, we click on **OK** and return to the Setup Data Source dialog. From the Setup Data Source dialog, we click **Ok** and return to the main dialog. We want to save our changes; so, we click the save registry toolbar button.




We are presented with the Save As dialog. We enter the name of the registry and click **OK**. We are through specifying a data source and exit the main dialog. The Registry Editor exits.




The registry is ready to use.

### *The Message Facility*

DDO uses common components. One such common component is the message facility. The message facility uses java.text.MessageFormat as the means of producing language-specific user messages, containing number, currency, percentages, date, time, and string variables. As a result, the message facility provides full Java message formatting facilities. Further, the message patterns are obtained from java.util.PropertyResourceBundle instances. A property resource bundle is a special type of resource bundle whose values are stored in a property file. To minimize message duplication, the message facility uses introspection to obtain inheritance and implementation graphs in locating a message pattern. The message facility provides message triggered diagnostic aids.

#### Locating Message Properties

All message property resource bundles are located in the *msgs* directory. The parent directory of the *msgs* directory is specified in the CLASSPATH. The java.util.ResourceBundle *getBundle()* method is used to load the message property resource bundles. The message facility uses a distinct naming scheme to insure that class name collisions do not occur. This scheme facilitates having the property resource bundles in one directory.

To retrieve a property resource bundle, two pieces of information are required: the fully qualified base name of the bundle and a locale identifier. These two strings are concatenated together, separated by an underscore to form a class name. An attempt is then made to load a class with that name using the default system loader. If a class with the name cannot be loaded, then the name is successively shortened until a resource bundle class is successfully loaded and instantiated. The *getBundle()* method will look for a property resource bundle whenever a class name fails to produce a resource bundle object. In particular, the class name is appended with the string ".properties". If such a file exists, a PropertyResourceBundle object is created for that properties file.

The naming scheme for the message facility is:

- All property resource bundles, i.e., message files, are placed in the *msgs* directory.
- The directory containing the *msgs* directory is in the CLASSPATH.

- The separators ('.') of the fully qualified class name, i.e., <package name>.<class name>, are changed to underscores ('_') to produce a file name.
- The file name extension, ".properties" is used to complete the file name.

For example, the message file name for the class com.sqribe.access.Access would be com_sqribe_access_Access.properties. This file would be placed in the *msgs* directory.

An application may be composed of hundreds or thousands of classes. Many classes could share the same messages. It would be tedious to have each class reflect its messages in a its own message file. Indeed, this would not reflect class relationships within the application. Java encourages the use of packages. Packages can be viewed as autonomous components. The classes within a package form a close knit set of relationships. These relationships are often exposed through inheritance and interface relationships. Further, component users also extend classes or implement interfaces prescribed in a package. The message facility recognizes these relationships and uses them to locate message properties. For example:

- The DDO relational database driver is in the package com.sqribe.jdbcacc.
- Within this package, com.sqribe.jdbcacc.JDBCacc is an empty interface, extending the interface com.sqribe.access.Access.
- Other classes in the com.sqribe.jdbcacc package implement the JDBCacc interface.
- The com.sqribe.access.Access interface extends the com.sqribe.comutil.Util interface. Like the JDBCacc interface, these are also empty interfaces.

Given a class, say com.sqribe.jdbcacc.JDBCConnection, in the driver, the message facility will fabricate a message file name from this fully qualified class name. The fabricated name is given to the resource bundle to locate a message property name. If the message property name was not found, the message facility follows the implementation graph repeating the query. If the query is still not satisfied, then the facility follows the inheritance graph for the class, repeating the query. Let's follow the search for the message property, "SchemaException.text".

1. Look in com_sqribe_jdbcacc_JDBCConnection*.properties.
2. Look in com_sqribe_jdbcacc_JDBCacc*.properties.

The asterisk ('*') in the name indicates where the resource bundle search appends the localization suffix. Since message property files only exist for the packages, only one message property files is searched.

1. com_sqribe_jdbcacc_JDBCacc*.properties.

Had the message property been, "DescriptionNotDefined.text", then the search would have continued and these message property files would have been included in the search:

2. com_sqribe_access_Access*.properties.
3. com_sqribe_comutil_Util*.properties.

Let's look at the affect of adding a French localization (without the country identifier) to the message property search. For this localization, we have translated the user message properties, e.g., "SchemaException.text", but not the log message properties, e.g., "SchemaException.logtext". When we search for "SchemaException.text", the message property file searched is:

1. com_sqribe_jdbcacc_JDBCacc_fr.properties.

On the other hand when we search for "SchemaException.logtext", the message property files searched are:

1. com_sqribe_jdbcacc_JDBCacc_fr.properties.
2. com_sqribe_jdbcacc_JDBCacc.properties.

The French message property file extends the default message property file, supplying in this case localized user message patterns. This algorithm is different from simple java.util.ResourceBundle processing, where a localized resource bundle represents a complete replacement of the less specific resource bundle. Here, the localized resource bundle extends the less specific bundle.

Message property files are Java property resource bundles. Property resource bundles are Java Properties. A property consists of a property name and a property value. The message facility interprets a property name as a message identifier plus a message type extension. In the following table, the message identifier is indicated by *msg*. The identifier may constitute a hierarchical name.

| Name | Value | Description |
|---|---|---|
| *Msg*.stri | Localized | *String* indicates a localized string |

| ng | message label | constant. |
|---|---|---|
| *Msg*.text | Localized message format text | *Text* indicates a message requiring formatting, while *string* indicates a localized string constant (that may be used as a substitution in a formatted message). |
| *Msg*.logtext | Localized log message format text | *Logtext* indicates a log message requiring formatting. |
| *Msg*.diagnostic | Fully qualified class name | *Diagnostic* is used to instantiate a class to perform diagnostics as a consequence of the given message. |
| *Msg*.exception | Fully qualified class name | *Exception* is used to create an Exception object to be thrown by the message facility. |

**Message Text**

The message facility supports three kinds of message patterns.

**String**

A string is a message text having no substitution elements. A string may be used as a substitution element or may appear in a user interface as a resource. The Java message format class is not used to process message strings. These are likely candidates for localization.

**Text**

A text message is a message pattern used for a user message or the string returned from java.lang.Throwable *getMessage()*. The Java message format class is used to process substitution elements appearing in the message pattern. . These are likely candidates for localization.

**LogText**

A logtext message is a message pattern used for application log messages. These message patterns contain detailed information that may be used for security, diagnostics, or support. The Java message format class is used to process substitution

elements appearing in the message pattern. . These are not likely candidates for localization.

**Services**

In addition to message processing, the message facility offers diagnostic support. The support is triggered when a message is written. The support tags are:

**Diagnostic**

This tag is used to instantiate a class to perform diagnostics related to the cause of the message. This may be used in a production or debugging mode to capture information affiliated with a message, where insufficient log information would be available. The value of this property is the fully qualified class name of the diagnostic object. The class must have a public constructor of the form: classname(String pMsgid, Object[] pSubs). The constructor should perform the diagnostics and cleanup its resources. The message facility does not retain a reference to the instantiated diagnostic class. You would use this feature to report supplemental diagnostic information as the result of a message and continue the normal processing flow for the message. This kind of support is useful when the condition causing the message is intermittent, environment specific, or timing related. In such cases it may be impractical to run an application trace or change the executable.

**Exception**

This tag is used to throw an exception for the given class name. This may be used to drive a specific form of error recovery. The compiler will not be able to detect the throw class. The class must have a constructor of the form: classname(String pMsgid, Object[] pSubs). You can use this tag to override the application behavior associated with a message. For example, you may simply want to obtain a stack trace and terminate the application. This can be accomplished by nesting a throw inside the exception constructor, catching the resulting exception, and use the *printStackTrace()* method to send the trace to System.err. Subsequently, the message facility will throw this exception.